Titre: Title:	Reducing Object-Oriented Testing Cost Through the Analysis of Antipatterns
Auteur: Author:	Aminata Sabane
Date:	2015
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Sabane, A. (2015). Reducing Object-Oriented Testing Cost Through the Analysis of Antipatterns [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1985/

UNIVERSITÉ DE MONTRÉAL

REDUCING OBJECT-ORIENTED TESTING COST THROUGH THE ANALYSIS OF ANTIPATTERNS

AMINATA SABANE
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
DÉCEMBRE 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

REDUCING OBJECT-ORIENTED TESTING COST THROUGH THE ANALYSIS OF ANTIPATTERNS

présentée par : SABANE Aminata
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. MULLINS John, Ph. D., président
M. GUÉHÉNEUC Yann-Gaël, Doctorat, membre et directeur de recherche
M. ANTONIOL Giuliano, Ph. D., membre et codirecteur de recherche
M. BAUDRY Benoit, Doctorat, membre
M. CECCATO Mariano, Ph. D., membre externe

To my parents
To my husband
To my brothers and sisters.

ACKNOWLEDGEMENTS

Many people contributed to the achievement of this research work and I would like to express my deep gratitude to all of them.

Firstly, I'm very grateful to my supervisors, Professors Yann-Gaël Guéhéneuc and Giuliano Antoniol, for their continuous support throughout my thesis, their availability, and their patience. Their difference of styles was sometimes disconcerting but most of the time rewarding. They shared with me their immense knowledge in software engineering and how to perform research. Thanks for their awesome advices for my research work and their tireless encouragement that were crucial for the completion of this work.

Besides my supervisors, I would like to thank Dr. Massimiliano Di Penta and Dr. Philippe Galinier. It was a great privilege for me to collaborate with you. Thanks for sharing your expertise with me and helping me to reach my goals.

Special thanks to my fellow labmates for the friendship, the availability, and the collaboration. Thanks for the constructive comments and also for all the fun we have even in stressful moments.

My sincere gratitude goes to my family : my parents Souba and Zenaba, my sisters Rokia and Salimata, and my brothers Idrissa and Abdoul Salam. Thanks for their love, their prayers, and their support during all these years. I'm endless indebted to them.

I am also very grateful to my beloved husband Issiaka for his love, his patience, his understanding, and his support. Thanks for encouraging me at each moment specially when my sky was dark.

I would also like to thank my lovely friend, Aya, for having read my thesis to help me improve it. Thank you for your availability, your affection, and your support.

My research work has been partially funded by Canadian International Development Agency (CIDA). I would like to thank them for giving me the opportunity to continue my studies.

I would like to end by thanking all my friends and everyone who has contributed in any way to the completion of this thesis. Find here the expression of my deep gratitude.

RÉSUMÉ

Les tests logiciels sont d'une importance capitale dans nos sociétés numériques. Le bon fonctionnement de la plupart des activités et services dépendent presqu'entièrement de la disponibilité et de la fiabilité des logiciels. Quoique coûteux, les tests logiciels demeurent le meilleur moyen pour assurer la disponibilité et la fiabilité des logiciels. Mais Les caractéristiques du paradigme orienté-objet—l'un des paradigmes les plus utilisés—complexifient les activités de tests. Cette thèse est une contribution à l'effort considérable que les chercheurs ont investi ces deux décennies afin de proposer des approches et des techniques qui réduisent les coûts de test des programmes orientés-objet et aussi augmentent leur efficacité.

Notre première contribution est une étude empirique qui vise à évaluer l'impact des antipatrons sur le coût des tests unitaires orienté-objet. Les antipatrons sont des mauvaises solutions à des problèmes récurrents de conception et d'implémentation. De nombreuses études empiriques ont montré l'impact négatif des antipatrons sur plusieurs attributs de qualité logicielle notamment la compréhension et la maintenance des programmes. D'autres études ont également révélé que les classes participant aux antipatrons sont plus sujettes aux changements et aux fautes. Néanmoins, aucune étude ne s'est penchée sur l'impact que ces antipatrons pourraient avoir sur les tests logiciels. Les résultats de notre étude montrent que les antipatrons ont également un effet négatif sur le coût des tests : les classes participants aux antipatrons requièrent plus de cas de test que les autres classes. De plus, bien que le test des antipatrons soit coûteux, l'étude révèle aussi que prioriser leur test contribuerait à détecter plutôt les fautes.

Notre seconde contribution se base sur les résultats de la première et propose une nouvelle approche au problème d'ordre d'intégration des classes. Ce problème est l'un des principaux défis du test d'intégration des classes. Plusieurs approches ont été proposées pour résoudre ce problème mais la plupart vise uniquement à réduire le coût des stubs quand l'approche que nous proposons vise la réduction du coût des stubs et l'augmentation de la détection précoce des fautes. Pour ce faire, nous priorisons les classes ayant une grande probabilité de défectuosité, comme celles participant aux antipatrons. L'évaluation empirique des performances de notre approche a montré son habilité à trouver des compromis entre les deux objectifs. Comparée aux approches existantes, elle peut donc aider les testeurs à trouver des ordres d'intégration qui augmentent la capacité de détection précoce des fautes tout en minimisant le coût de stubs à développer.

Dans notre troisième contribution, nous proposons d'analyser et améliorer l'utilisabilité de

Madum, une stratégie de test unitaire spécifique à l'orienté-objet. En effet, les caractéristiques inhérentes à l'orienté-objet ont rendu insuffisants les stratégies de test traditionnelles telles que les tests boîte blanche ou boîte noire. La stratégie Madum, l'une des stratégies proposées pour pallier cette insuffisance, se présente comme une bonne candidate à l'automatisation car elle ne requiert que le code source pour identifier les cas de tests. Automatiser Madum pourrait donc contribuer à mieux tester les classes en général et celles participant aux antipatrons en particulier tout en réduisant les coûts d'un tel test. Cependant, la stratégie de test Madum ne définit pas de critères de couverture. Les critères de couverture sont un préalable à l'automatisation et aussi à la bonne utilisation de la stratégie de test. De plus, l'analyse des fondements de cette stratégie nous montre que l'un des facteurs clés du coût des tests basés sur Madum est le nombre de "transformateurs" (méthodes modifiant la valeur d'un attribut donné). Pour réduire les coûts de tests et faciliter l'utilisation de Madum, nous proposons des restructurations du code qui visent à réduire le nombre de transformateurs et aussi des critères de couverture qui guideront l'identification des données nécessaires à l'utilisation de cette stratégie de test.

Ainsi, partant de la connaissance de l'impact des antipatrons sur les tests orientés-objet, nous contributions à réduire les côuts des tests unitaires et d'intégration.

ABSTRACT

Our modern society is highly computer dependent. Thus, the availability and the reliability of programs are crucial. Although expensive, software testing remains the primary means to ensure software availability and reliability. Unfortunately, the main features of the object-oriented paradigm (OO)—one of the most popular paradigms—complicate testing activities. This thesis is a contribution to the global effort to reduce OO software testing cost and to increase its reliability.

Our first contribution is an empirical study to gather evidence on the impact of antipatterns on OO unit testing. Antipatterns are recurring and poor design or implementation choices. Past and recent studies showed that antipatterns negatively impact many software quality attributes, such as maintenability and understandability. Other studies also report that antipatterns are more change- and defect-prone than other classes. However, our study is the first regarding the impact of antipatterns on the cost of OO unit testing. The results show that indeed antipatterns have a negative effect on OO unit testing cost: AP classes are in general more expensive to test than other classes. They also reveal that testing AP classes in priority may be cost-effective and may allow detecting most of the defects and early.

Our second contribution is a new approach to the problem of class integration test order (CITO) with the goals of minimizing the cost related to the order and increasing early defect detection. The CITO problem is one of the major problems when integrating classes in OO programs. Indeed, the order in which classes are tested during integration determines the cost (stubbing cost) but also the order on which defects are detected. Most approaches proposed to solve the CITO problem focus on minimizing the cost of stubs. In addition to this goal, our approach aims to increase early defect detection capability, which is one of the most important objectives in testing. Early defect detection allows detecting defects early and thus increases the cost-effectiveness of testing. An empirical study shows the superiority of our approach over existing approaches to provide balanced orders: orders that minimize stubbing cost while maximizing early defect detection.

In our third contribution, we analyze and improve the usability of Madum testing, one of the unit testing strategies proposed to overcome the limitations of traditional testing when testing OO programs. Opposite to other OO unit testing, Madum testing requires only the source code to identify test cases. Madum testing is thus a good candidate for automation, which is one of the best ways to reduce testing cost and increase reliability. Automatizing Madum testing can help to test thoroughly AP classes while reducing the

testing cost. However, Madum testing does not define coverage criteria that are a prerequisite for using the strategy and also automatically generating test data. Moreover, one of the key factors in the cost of using Madum testing is the number of transformers (methods that modify a given attribute). To reduce testing cost and increase the easiness of using Madum testing, we propose refactoring actions to reduce the number of transformers and formal coverage criteria to guide in generating Madum test data. We also formulate the problem of generating test data for Madum testing as a search-based problem.

Thus, based on the evidence we gathered from the impact of antipatterns on OO testing, we reduce the cost of OO unit and integration testing.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

LIST OF ACRONYMS AND ABBREVIATIONS

AP(s)	Antipattern(s)
APDF	Average Percentage of Faults Detected
CITO	Class integration Test Order
EA	Evolutionary Algorithm
ECG	Enhanced Call-Graph
EDDR	Early Defect Detection Rate
DECOR	Defect dEtection for CORrection
GA	Genetic Algorithm
HC	Hill Climbing
MA	Memetic Algorithm
Madum	Minimal Data Members Usage Matrix
MITER	Minimizing Integration Testing EffoRt
NEDDR	Normalized Early Defect Detection Rate
ORD	Object Relation Diagram
SBC	Server-Before-Client
SBSE	Search-Based Software Engineering
SBST	Search-Based Software Testing

CHAPTER 1 INTRODUCTION

1.1 Context and Motivation

Software are prevalent in our daily life and their reliability is crucial. The proper functioning of most of activities in our modern societies relies heavily on software availability and reliability. Moreover, in many critical areas that use software, such as aviation and medicine, software failures may cause economic loss and–or human casualties. Testing is to date the primary and the most used means to ensure software availability and reliability. However, software testing is an expensive and time consuming activity: literature reports that testing activities cost up to 50% of software overall cost (Beizer, 1990; Ellims *et al.*, 2006). Moreover, testing activities have been complicated by object-oriented (OO) paradigm that is nevertheless the most popular software development paradigm.

First OO languages have been created in 70s but the paradigm has been widely accepted in 90s with the Smalltalk, C++, and Java languages. OO paradigm is a development paradigm based on the encapsulation of state and behavior as opposite to procedural programming that advocates a centralization with a functional decomposition (Riel, 1996). It has been created to overcome the limitations of procedural programming languages when building complex and large software. Although OO is no the silver bullet (Brooks, 1987), it improves reusability, flexibility, and it facilitates building large and complex programs (Booch, 1991; Binder, 1999).

However, OO does not only change the way of designing and developing programs, it impacts and complicates the way of testing them (Binder, 1999; Bashir et Goel, 2000). Indeed, the core features of OO—abstraction, encapsulation, visibility, inheritance, and polymorphism—that are responsible for its success are the same that make difficult OO testing. For example, encapsulation and visibility prevent to directly access the state of object during test and, thus, force to find different ways to ascertain the state of the object: use of friend mechanism in C++, use of accessors in Java, or reflection in both. Another example concerns the shift of the unit from *function* in procedural programming to *class* in OO. This shift makes traditional unit testing strategies, such as black box and white box, insufficient regarding the test of OO (Binder, 1999; Bashir et Goel, 2000). Thus, testing a single method in terms of inputs/outputs is no longer enough. OO unit testing requires to consider the state of the object during the test and also interactions between methods (Bashir et Goel, 2000). Because they are strongly structure dependant, unit testing and integration testing are the most impacted testing levels by OO. Consequently, many research works have been carried

out to meet the challenges of testing OO programs, reduce the cost, and increase the efficiency of testing.

Researchers proposed various testing strategies and criteria that consider the particularities of OO programs and overcome the limitations of traditional testing strategies (Binder, 1999; Bashir et Goel, 2000; Briand *et al.*, 2003b). They strived to find ways to promote OO testing, automatically generate test data (Tonella, 2004; McMinn et Holcombe, 2005; Sakti *et al.*, 2015; Fraser *et al.*, 2015) as well as to identify factors impairing OO testing cost and effectiveness (Jungmayr, 2002; Bruntink et van Deursen, 2006; Badri *et al.*, 2010). Indeed, the knowledge about the factors that impede the cost of testing or its efficiency is essential to take adequate actions and propose solutions to reduce that cost and increase the effectiveness.

Antipatterns (APs) are defined as recurring and poor design or implementation choices (Brown *et al.*, 1998). The time-to-market, lack of understanding, and misuse of OO concepts are among the main factors that lead developers to introduce antipatterns in their code. Many empirical studies have been performed to assess the impact of antipatterns on software quality (Deligiannis *et al.*, 2003; Du Bois *et al.*, 2006; Khomh *et al.*, 2012). The results of these studies show that the presence of antipatterns in programs negatively impact many software quality attributes, such as maintenability (Deligiannis *et al.*, 2003) and understandability (Abbes *et al.*, 2011). Moreover, other studies reveal that classes involving antipatterns are more change- and defect-prone than other classes (Olbrich *et al.*, 2009; Khomh *et al.*, 2012). Thus, it is important to investigate on the potential impact of antipatterns on OO testing. The outcome of such a study can serve to propose approaches to improve OO unit and integration testing and reduce their cost. However, to the best of our knowledge, there is no empirical evidence about the potential impact of antipatterns on OO testing.

1.2 Thesis Statement and Contributions

The considerations discussed in the previous section motivate us to formulate our thesis statement as follows.

> The cost of testing classes involving antipatterns is higher than that of other classes but it is possible to offer techniques to reduce that cost during unit and integration testing.

In summary, to support our thesis, we first study the impact of APs on the cost of OO unit (class) testing. Based on the results of this study, we propose a new approach to the class integration test order (CITO) that balances the cost of stubs and early defect detection. Besides reducing the stubbing cost as existing approaches to the CITO problem,

the proposed approach aims also to increase early defect detection capability. Finally, we analyze and improve the usability of Madum testing, a specific OO unit testing strategy proposed to overcome the limitations of traditional unit testing when testing OO programs. This investigation aims at contributing to the effort of finding suitable testing strategies to test OO programs containing APs.

1.2.1 Contribution 1: Impact of Antipatterns on Class Testing

The literature reports numerous studies on the impact of antipatterns (APs) on many software quality attributes. The results of these studies point out the negative impact of APs on software quality: APs decrease software maintenability (Deligiannis *et al.*, 2003) and understandability (Abbes *et al.*, 2011) and thus increase software maintenance cost. Other studies also showed that classes involving APs are less resilient to change and defect (Olbrich *et al.*, 2009; Khomh *et al.*, 2012). Although testing is one of the most expensive activities, to the best of our knowledge, there is no study that evaluates the potential impact of APs on testing. These considerations motivate us to study and gather evidence on the impact of APs on OO testing. Such evidence is important for researchers and practitioners. Evidence on the negative impact of APs on testing will allow researchers to investigate new research directions to improve OO testing when taking into account the presence of APs. Based on this evidence, practitioners could take informed decisions about how to deal with APs in their programs: they could knowingly choose to keep them, remove them, or refactor them.

To perform our study, we use as measure of testing cost the number of test cases required by Madum testing, a unit testing strategy specific to OO programs. The results of the study show that APs (and specially some APs, such as Blob, AntiSingleton, ComplexClass, or SwissArmyKnife) impact negatively unit (class) testing. Indeed, classes participating to APs require more test cases than other classes. The results also reveal that although APs testing is expensive, its prioritization is cost-effective in the sense that it allows detecting most of the defects and early.

1.2.2 Contribution 2: MITER (Minimizing Integration Testing EffoRt)

One of the main problems faced when testing OO programs during integration is the order in which classes should be tested (Kung *et al.*, 1995). Indeed, this order highly impacts the cost of integration testing as it determines the number and–or the complexity of stubs to be developed (Tai et Daniels, 1997). Each time a class not developed or not tested yet is required to test another class, a stub should be developed to mimic its behavior during the

test [1]. As stated by Kung *et al.* (1995), the cost of stubs constitutes a major factor in the overall cost of class integration testing: developing a stub to replace a class is more complex and difficult than developing one to replace a function in procedural programming.

This problem, known as class integration test order (CITO) problem, has been widely investigated (Kung *et al.*, 1995; Tai et Daniels, 1997; Traon *et al.*, 2000; Briand *et al.*, 2002b; Abdurazik et Offutt, 2009) with the goal of reducing the stubbing cost. Besides the stubbing cost, the order in which classes are tested defines also the order in which defects will be detected. Thus, this order also impacts the defect detection of the testing activity. Early defect detection is one of the major goals and most desirable properties of testing activities and its economical value is recognized and documented in the literature (Boehm et Papaccio, 1988; Dabney *et al.*, 2006; Peters *et al.*, 2013). Early defect detection allows developers to begin locating and correcting defects earlier. Thus, it makes the delivery of a release of better quality on time. It can also ensure that a maximum number of defects have been discovered even if testing gets stopped prematurely (Elbaum *et al.*, 2000). Moreover, as shown by the first contribution of this dissertation, although AP classes are expensive to test, prioritizing their test allows detecting most of the defects and early.

We propose MITER (Minimizing Integration Testing EffoRt), a new approach to the CITO problem, that seeks not only to minimize the stubbing cost but also to increase early defect detection when testing OO classes during integration. This novel approach relies on a Memetic Algorithm (MA) to compute class integration test orders. An evaluation of MITER shows its superiority to foster early defect detection while keeping low stubbing cost compared to existing approaches.

1.2.3 Contribution 3: Improvement of the Usability of Madum Testing

Previous studies show that AP classes are more defect-prone than other classes (Khomh *et al.*, 2012), advocating the need to test them thoroughly. However, our first study shows that their test is expensive. Therefore, it is necessary to find means to reduce the cost of testing AP classes.

Madum testing is one of the OO unit strategies proposed to address the limitations of traditional unit testing in the test of OO programs (Bashir et Goel, 2000). In contrast to other OO unit testing strategies, such as state-based testing (Binder, 1999) and pre-and-post conditions (Boyapati *et al.*, 2002) that require specific inputs (respectively state charts or pre-and-post conditions specifications), Madum testing relies only on source code syntax and the notions

1. This rule is called in this dissertation server-before-client (SBC) principle. It is defined in Section 2.1.4. Consequently, we use stub and SBC violation interchangeably.

of definition and use of attributes to identify test cases. Madum testing is thus a good candidate for automation which is one of the main means to reduce testing costs and increase testing use and efficiency: The automation of testing activities significantly reduces testing cost and helps also minimize human error and make regression testing easier (Ammann et Offutt, 2008). Automatize Madum testing could thus help reduce the test of AP classes. However, there is no formal criteria to guide in the use of this strategy and its automation neither evidence about its effectiveness and usability in practice. Moreover, Madum testing is based on the test of sequences of methods that modify values of attributes; these methods are called *transformers*. The number of these methods is then a key factor in the cost of the test. Therefore, our third contribution aims at addressing these issues. On a set of classes exhibiting a high number of transformers, we show that specific refactoring actions can indeed reduce the number of transformers and thus the cost of using Madum testing. We also propose formal criteria to guide in generating Madum test data. Finally, we formulate the problem of generating test data for Madum testing as a search-based problem.

1.3 Publications

Part of this research work has been the object of the following publications:

Aminata Sabané, Giuliano Antoniol, Philippe Galinier, Yann-Gaël Guéhéneuc, Massimiliano Di Penta (2015). MITER: Minimizing Integration Testing Effort. Journal of Software Maintenance and Evolution: Research and Practice (submitted).

Aminata Sabané, Giuliano Antoniol, Massimiliano Di Penta, Yann-Gaël Guéhéneuc (2013). A Study on the Relation Between Antipatterns and the Cost of Class Unit Testing. Proceedings of the 17th European Conference On Software Maintenance and Reengineering (CSMR).

Aminata Sabané (2010). Improving System Testability and Testing with Microarchitectures. Proceedings of the 17th Working Conference on Reverse Engineering (WCRE).

1.4 Roadmap

The reminder of this dissertation is organized as follows:

Chapter 2 (p. 7): This chapter provides background materials to facilitate the understanding of this dissertation. It is organized under the main topics related to our thesis statement: testing in general and OO unit and integration testing in particular, antipatterns, search-based techniques, defect prediction, and empirical studies.

Chapter 3 (p. 23): This chapter reports the main related work to this dissertation in-

cluding antipatterns, their impact on software quality, and the main approaches to the class integration test order problem.

Chapter 4 (p. 28): This chapter presents the study on the impact of antipatterns on the cost of class testing.

Chapter 5 (p. 42): This chapter presents our new approach, MITER, to the problem of class integration test order with the goals of minimizing stubbing cost and increasing early defect detection. It also reports the experiment we performed to assess the performance of MITER to find a trade-off between the two objectives.

Chapter 6 (p. 81): This chapter analyzes the usability of Madum testing. It proposes specific refactoring actions to reduce the cost of using this strategy, defines formal coverage criteria to guide in generating test data, and formulates the problem of generating Madum test data as a search-based problem.

Chapter 7 (p. 91): This chapter summarizes the contributions of this thesis and outlines possible future directions.

Appendix A (p. 104): This Appendix presents the source code of the class *BankAccount* used to illustrate Madum testing.

Appendix B (p. 107): This Appendix presents an experiment performed to compare the performance of genetic algorithm and memetic algorithm in finding close to optimal orders when solving the problem of class integration test order.

CHAPTER 2 BACKGROUND

As described in the previous chapter, the main topics addressed in our thesis work are testing, antipatterns, search-based techniques, defect prediction, and empirical studies. To facilitate the understanding of the following chapters, the present chapter defines the basic and main concepts related to these topics. It also presents the main techniques and tools used in this dissertation.

2.1 Concepts Related to Testing

Software testing consists in the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the generally infinite execution domain (Bourque et Fairley, 2014). Glenford *et al.* (2004) highlight the primary purpose of software testing: "software testing is a process of executing a program or application with the intent of finding defects." Although testing cannot guarantee the absence of defects (Dahl *et al.*, 1972), testing is one of the best means to ensure software dependability. As mentioned in the previous chapter, the OO paradigm affects testing process and strategies in particular at unit and integration levels. In the following, we recall basic testing concepts based on the book of Ammann et Offutt (2008). We also describe unit and integration testing in the context of OO programs. Finally, we present a specific OO unit testing strategy used in this dissertation namely Madum testing.

2.1.1 Basic Testing Concepts

Levels of Testing: Testing levels are defined based on traditional software process steps. Thus, each development activity is associated to a level of testing leading to five main levels of testing. A given testing level aims at discovering specific defects and usually defects introduced in the corresponding development phase. The information used in a given testing level is derived also from the output of the related development phase. This categorization does not imply the use of a waterfall development process and can be used in any development process. We briefly describe each testing level in the following.

– *Unit Testing* is associated with the implementation phase and is used to verify that each unit works correctly in isolation. A unit is defined as a software component that cannot be subdivided into other components. A unit is a function or procedure in procedural programs whereas it is a class in OO programs.

– *Module Testing* corresponds to the detailed design phase. It assesses individual modules where a module represents a collection of related units.
– *Integration Testing* is associated to the subsystem design phase. It refers to testing activities used to evaluate the interactions between modules.
– *System Testing* is related to the architectural design phase and aims at determining if the software works as a whole.
– *Acceptance Testing* corresponds to the requirements analysis phase. Its goal is to assess whether the completed software meets the customers' needs as specified in the requirements.

Test Requirements: A test requirement is a specific element of a program artefact that a test case must satisfy or cover. All statements or all branches are examples of test requirements. It happens that some test requirements cannot be satisfied; those test requirements are called *infeasible test requirements*.

Coverage Criterion: A coverage criterion is a rule or collection of rules that impose test requirements on a test set. For example, the branch coverage criterion requires to cover all branches in the program under test. Testing criteria are very important in testing: they guide testers on what must be tested, provide measurements of test quality, and they are also stopping criteria that determine whether sufficient testing has been done (Zhu *et al.*, 1997).

Coverage Level: Coverage level or simply coverage with respect to a given criterion is the ratio of the number of test requirements satisfied out of the total number of test requirements (Ammann et Offutt, 2008). Coverage level is one of the metrics used to assess the quality or the degree of testing. Although, it has been shown to not directly impact testing effectiveness (Briand et Pfahl, 1999; Inozemtseva et Holmes, 2014), it is a necessary condition for testing effectiveness. Indeed, a tester cannot discover a defect if the test requirement that can reveal that defect is not satisfied. For example, if a bloc contains a defect and is not executed, the defect will not be discovered. Another example is the case where a test requirement with a specific category of inputs in category partition testing leads to a failure. If any test case is designed to test this category, the failure will not happen and the defect will not be discovered.

Criteria Subsumption: Coverage criteria are often related to each other. This relationship is expressed as the criteria subsumption: a coverage criterion $C1$ subsumes a coverage criterion $C2$ if and only if satisfying $C1$ will guarantee that $C2$ is satisfied. For example, branch coverage criterion subsumes statement coverage criterion.

2.1.2 OO Unit Testing

Unit testing consists in the test of basic components of the program in isolation. It is the lowest level in testing process. Encapsulation—one of the main features of OO programming—changes the definition of a unit as known in procedural programming. This change represents one of the major changes in OO testing concerns (Bashir et Goel, 2000). In OO programs, the unit is no longer a function as in procedural programs but an object (instance) of the class under test. An object encapsulates its state and its behavior (associated methods). Therefore, the output of a method does not only depend on the inputs but also on the state of the object on which it has been called. This change of the unit and the notion of object state lead to the emergence of a new testing level and of new testing techniques and approaches (Beizer, 1990; Bashir et Goel, 2000). OO unit testing involves the following steps (Ammann et Offutt, 2008):

– *Intramethod testing* consists in the test of individual methods.
– *Intermethod testing* consists in the test of pairs of methods within a same class.
– *Intraclass testing* corresponds to the test of a class as unit and consists in the test of sequences of methods of that class on some of its instances.

The two first levels are similar to the unit and module testing in procedural programming whereas the third one is specific to OO programming. In OO programs, defects may depend on the behavior of particular methods when the class is in a certain state.The intraclass testing has for goal to detect such defects by identifying the sequence of methods that will put an object under test in an abnormal state (Bashir et Goel, 2000; Ammann et Offutt, 2008). As pointed out by many authors (Binder, 1999; Bashir et Goel, 2000), the existing procedural black-box and white box unit testing techniques are insufficient for intraclass testing. Indeed, they are conceived to test functions as stand-alone and therefore could miss state-based errors due to interactions between methods. To overcome this limitation, new techniques, such as state-based testing (Binder, 1999), pre-and-post conditions testing (Boyapati *et al.*, 2002), and Madum testing (Bashir et Goel, 2000) have been proposed. Because the number of sequences of methods grows exponentially with the number of methods within a class, testing all possible sequences of methods is very expensive if not impossible (Bashir et Goel, 2000). Indeed, the number of possible sequences in a class with five methods is 120 whereas it is 5,040 for seven methods and 3,628,800 for just 10 methods when considering sequences that will contain each method once. OO unit testing strategies, based on specific criteria, help to select only a subset of methods sequences to test; thus they reduce the number of sequences to test within a class while keeping an acceptable degree of confidence in the test.

2.1.3 Madum Testing

Madum testing, proposed by Bashir et Goel (2000), is one of the class testing strategies designed to address the limitations of procedural unit testing strategies regarding OO unit testing. Madum testing helps perform intraclass testing by focusing only on testing methods that modify the state of the object. It thus reduces the number of possible sequences of methods to test. The approach is based on *data slices*, defined as the set of methods that access or modify a field (attribute). The correctness of a class is tested in terms of the correctness of all its slices tested separately.

The identification of each slice is based on two key elements: the enhanced call-graph (ECG) and the minimal data members usage matrix (Madum). An ECG represents the accesses (usages or invocations) of members of a class by other members. The ECG of a class C can be defined as: $ECG(C) = (M(C), F(C), E_{mf}, E_{mm})$, where:
- $M(C)$ represents the set of methods of C;
- $F(C)$ is the set of fields of C;
- $E_{mf} = \{(m_i, f_j)\}$ indicates that there is an edge between the method m_i and the field f_j, *i.e.*, m_i accesses the field f_j;
- and $E_{mm} = \{(m_i, m_j)\}$ indicates that there is an edge between the methods m_i and m_j, *i.e.*, m_i invokes m_j.

Madum testing uses a taxonomy that classifies methods into four categories:

- Constructors (c): class constructors;
- Transformers (t): methods that alter the state of one or more fields;
- Reporters (r): methods that return the value of a field;
- Others (o): methods that do not fit in the three first categories, *e.g.*, methods that handle special conditions/exceptional behavior.

We can consider a constructor as a specific transformer. It is a transformer because it initializes therefore changes the value of attributes. It is specific because its position when considering sequences of methods does not change, i.e, always first.

A Madum is an $n_f \cdot n_m$ matrix, where n_f and n_m are respectively the number of fields and the number of methods of the class. Built based on the ECG of the class, each cell (i, j) of the Madum is marked as follows:
- c if m_j is a constructor that initializes f_i;
- t if m_j transforms f_i;
- r if m_j reports the state of f_i;
- o if m_j accesses to f_i without being a constructor, a transformer, or a reporter.

A method m_j can access a field f_i directly or indirectly through another method m_k (transitively) invoked by m_j. However, if m_j accesses f_i only through m_k, and m_k has been tested already in the f_i slice, then according to the strategy, there is no need to also test m_j in the f_i slice.

Madum testing works as follows. First, reporters are tested, followed by setters (if present) and by constructors. Then, interactions among transformers are tested. These interactions are based on the permutation of slice transformers for each constructor context. Let c be the set of constructors and t be the set of transformers in a given slice. Then, to test sequences of transformers in that slice, the tester must produce $|c| \cdot |t|!$ test cases, where $|c|$ is the number of constructors and $|t|$ the number of transformers in the slice. Finally, the others (o) are tested using traditional black or white-box testing strategies.

The process to test interactions among transformers implies that the number of sequences to test in a slice will grow exponentially with its number of transformers.

An Example of Madum

We illustrate the different concepts of Madum testing using a class BankAccount in Appendix A as example. BankAccount is a simple class representing a checking account. It has three private and non-static attributes (*name*, *accountNumber*, and *balance*), two constructors (*BankAccount(String)* and *BankAccount(String, double)*), and seven public and non-static methods (*deposit(double)*, *getAccountId()*, *getBalance()*, *getName() payInterest()*, *printAccount()*, and *withdraw(double)*). Figures 2.1 and 2.2 represent respectively the ECG and the Madum of the class BankAccount. The attribute balance has three transformers and the other attributes do not have any. The method *printAccount()* is marked as *other* because it only displays the attributes values. Because the constructor *BankAccount(String)* initializes the attributes through the second constructor *BankAccount(String, double)*, it does not need to be tested and thus it does not appear in any slice.

2.1.4 OO Integration Testing

Integration testing consists in assessing interactions between two components that have already been tested in isolation during unit testing (Beizer, 1990). The main purpose of this testing level is to ensure that components that work in isolation communicate correctly. In OO programming, the components to integrate can be classes, aggregations of classes, or subsystems.

To perform integration testing, one should answer two main questions (Briand *et al.*, 2003b):

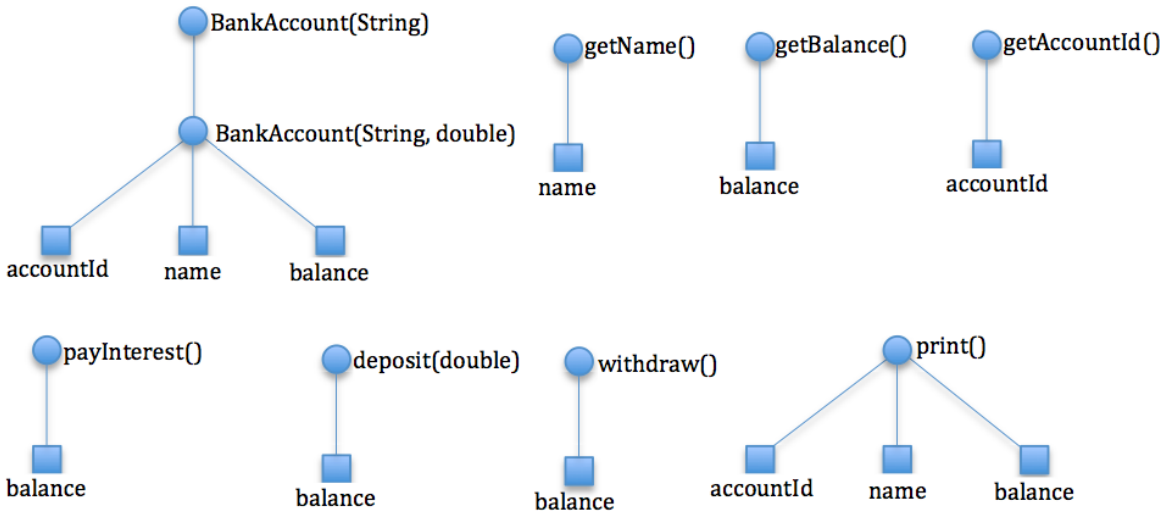


Figure 2.1 BankAccount Enhanced Call-Graph.

	BankAccount (String)	BankAccount (String,double)	getAccountId()	getName()	getBalance()	payInterest()	Deposit (double)	Withdraw (double)	Print()
accountId		c	r						o
name		c		r					o
balance		c			r	t	t	t	o

Figure 2.2 BankAccount Minimal Data Members Usage Matrix.

– How to test the interactions between two components?
– In which order components should be integrated?

Many approaches have been proposed to answer the first question (Bashir et Goel, 2000; Briand *et al.*, 2003b). The second question which is of interest in this dissertation has also been investigated through the well known class integration test order (CITO) problem when the components to integrate are classes (Kung *et al.*, 1995; Traon *et al.*, 2000; Abdurazik et Offutt, 2009). The CITO problem is defined as the problem of finding an order in which classes can be integrated and tested with minimum cost. The cost is usually expressed as the number and–or the complexity of stubs.

We present in the following the main concepts related to class integration test order.

Server and Client classes: A *server class* is a class that provides services to other classes whereas a *client class* is a class that receives services from others. A class can play both roles

and thus being a server class for certain classes and a client class for others.

Stubs: A *stub* is a dummy class used to simulate the behavior of a class that is not tested or developed yet when testing one of its clients (Traon *et al.*, 2000). We can distinguish two types of stubs:

– A *specific stub* simulates the services for the use of a given client only. In that case, the stub is specific to a particular client and as many specific stubs have to be created as there are clients (Traon *et al.*, 2000).
– A *realistic stub* simulates all services that the original class can provide. In that case, the stub works whatever the client (Traon *et al.*, 2000).

Server-before-client Principle: Most of the proposed approaches to the CITO problem (Kung *et al.*, 1995; Tai et Daniels, 1997; Traon *et al.*, 2000; Briand *et al.*, 2002b; Abdurazik et Offutt, 2009) are based on the principle that a server class should be tested before its clients to reduce integration testing effort. This principle is called *server-before-client (SBC) principle* in this dissertation. In theory, each time a client is tested before its server classes, there are SBC violations: stubs should be written to replace the server classes to avoid the consequences of using untested classes to evaluate the correctness of other classes: risk of defect propagation (Lloyd et Malloy, 2005) and increase of the effort required to locate a bug.

2.2 Antipatterns

Antipatterns are recurring, poor design choices that negatively impact software quality (Deligiannis *et al.*, 2004; Du Bois *et al.*, 2006; Olbrich *et al.*, 2009; Khomh *et al.*, 2012). One of the goals of this thesis is to study their impact on software testing. For that purpose, we choose to analyse 13 antipatterns (Brown *et al.*, 1998; Fowler, 1999). These antipatterns, described in Table 2.1, have been studied in previous research work (Khomh *et al.*, 2012; Romano *et al.*, 2012). They also appear frequently in the analyzed programs and they are representative of design and implementation problems with data, complexity, size, and the features provided by classes (Romano *et al.*, 2012).

To detect the occurrences of antipatterns in the analyzed programs, we rely on DECOR (Defect dEtection for CORrection) (Moha *et al.*, 2010). DECOR is an approach based on the automatic generation of detection algorithms from rule cards. It uses a domain-specific language to describe antipatterns in the form of rule cards. Rule cards are then automatically converted into detection algorithms through the framework DETEX.

2.3 Search-Based Techniques

Search-Based Software Engineering (SBSE) is a software engineering research area in which software engineering problems are reformulated as search-based problems and resolved using metaheuristic techniques (Harman *et al.*, 2012). Many problems in software engineering can be formulated as search-based problems, making popular the use of metaheuristic techniques in a wide range of software engineering problem domains, such as testing, design, requirements, project management, and refactoring (Harman *et al.*, 2012). Resolving a search-based problem results in writing a fitness function that will automatically guide a search in a large search space to find an optimal or nearly optimal solution.

The sub-area of SBSE concerned with software testing problems is called Search-Based Software Testing (SBST). SBST has attracted the interest of the testing community as shown by the number of publications in that area: almost half of all publications in SBSE are in SBST (Harman *et al.*, 2015). Many surveys on SBST literature have also been published (McMinn, 2004; Ali *et al.*, 2010; Yoo et Harman, 2012). Search-based techniques have been successfully used in many testing problems, such as test cases prioritization, class integration test ordering, and test data generation (Harman *et al.*, 2012). Search-based techniques, also called metaheuristic-search techniques, are high-level frameworks. They utilize heuristics to find solutions for combinatorial problems at a reasonable computational cost (McMinn, 2004). They can be easily adapted to solve optimization problems and are suitable for problems classified as NP or NP-hard. They are also useful to resolve problems for which a polynomial time algorithm is known to exist but is not practical (Harman *et al.*, 2012).

In the following, we define the concept of fitness function and its role in search-based techniques and move to a brief description of search-based techniques used in this dissertation.

2.3.1 Fitness Function

Fitness function, also called the objective function, is one of the key elements of search-based techniques, the other one being the representation of the problem (Harman *et al.*, 2012).

A fitness function is a characterization of what is considered to be a good solution (Harman et Jones, 2001). It guides the search by evaluating the quality of each candidate solution with respect to the overall search goal (Harman *et al.*, 2012).

2.3.2 Random Search Algorithm

Random search is the simplest search-based algorithm. It is the only one that does not use a fitness function (Harman *et al.*, 2012). Although random search is simple to implement, it often fails to find optimal solutions because it does not provide any guidance and the search is totally blinded. Generally, random search algorithms are used to produce solutions that serve as baselines to evaluate the performance of other search-based techniques.

2.3.3 Hill Climbing

Hill Climbing is a local search algorithm, one of the main families of search algorithms. With hill climbing, the search starts with the selection of a random point in the search space. It then examines the neighborhood of that point. The neighborhood is specific to the problem and represents the candidate solutions that are close or are similar to the current point (Harman *et al.*, 2012). If the fitness of one of the neighbors is better than that of the current point, the search moves to that point and the process is repeated until there is no candidate solution in the neighborhood of the current candidate solution that improves the fitness. Such a solution is said to be locally optimal and may not represent a globally optimal solution. To leave a local optimal and improve the performance of the algorithm, the hill climbing procedure can be restarted many times from different initial random points. The main elements in a hill climbing algorithm are the fitness function to evaluate each candidate solution, the operator (the move) used to identify the neighborhood, and the type of ascent strategy used to explore the neighborhood. Types of ascent strategy include steepest ascent, with all neighbors are evaluated and the move is performed towards the candidate solution that offers the greatest improvement of the fitness value. Another type of ascent is the random or first ascent strategy in which neighboring candidate solutions are evaluated at random and the first neighbor that improves the fitness function is chosen for the move (Harman *et al.*, 2012). Figure 1 shows a pseudo-code of the hill climbing in the context of a minimization problem.

Algorithm 1 HC Pseudo-code — Adapted from (Harman *et al.*, 2012).

Select randomly a starting point s in the search space S
Repeat
  Select $s' \in N(s)$, neighborhood of s, such as $fitness(s') < fitness(s)$ according to the ascent strategy
  $s < -s'$
Until $fitness(s' \geq fitness(s), \forall s' \in N(s)$

2.3.4 Genetic Algorithm

Genetic algorithm (GA) is a evolutionary algorithm (EA) that applies biological principles of evolution to artificial systems. To solve an optimization problem, a GA evolves iteratively a set of potential solutions—referred to as population. A GA may be defined as an iterative procedure that searches for the best solution of a given problem among a population. The search starts from an initial population of individuals, often randomly generated. At each evolutionary step, individuals are evaluated using the fitness function. The fittest individuals will have the highest probability to reproduce. The evolution (*i.e.*, the generation of a new population) is performed using two operators: the crossover operator and the mutation operator. The crossover operator takes two individuals (the parents) in the current generation and recombines them to produce one or two new individuals (the offspring). The mutation operator applies a perturbation to the offspring to prevent convergence to local optima and to diversify the search into new areas of the search space. The iterative process stops when the global optimal has been found or some stop criteria, often specified as a maximal number of generations or a time limit, have been met. Many variants of GA exist depending on the parents selection strategy, the replacement strategy, or the evolution of the population size. The pseudo-code of the GA is given in Figure 2. Further details on GA can be found in a book by (Goldberg, 1989).

Algorithm 2 GA Pseudo-code — Adapted from (Harman *et al.*, 2012).

Randomly generate initial population P
Repeat
  Evaluate fitness of each individual in P
  Select parents according to selection mechanism
  Apply crossover on parents to create new offspring O
  Apply mutation on offspring in O
  Generate new population P' from P and O
  $P < -P'$
Until Stopping Condition Reached
Return the best individual found

2.3.5 Memetic Algorithm

Memetic algorithms (MA), also a type of EA, are an enhancement of GAs. A Memetic algorithm (MA) is a modification of a GA where a local search operator is used along with crossover and (possibly) mutation (Moscato *et al.*, 1989). This kind of hybridization has been

proved to be powerful and makes MA as one of the most powerful existing meta-heuristics (Moscato *et al.*, 1989; Hoos et Sttzle, 2004). Many researchers in SBSE have also shown the superiority of MA over GA (Arcuri et Yao, 2007; Fraser *et al.*, 2015).

Algorithm 3 MA Pseudo-code.

Randomly generate initial population P

Apply local search on each individual of P
Repeat
  Evaluate fitness of each individual in P
  Select parents according to selection mechanism
  Apply crossover on parents to create new offspring O
  Apply mutation on offspring in O
  Apply local search on offspring
  Generate new population P' from P and O
  $P < -P'$
Until Stopping Condition Reached
Return the best individual found

2.4 Defect Prediction

Defect prediction models have been intensively studied in software engineering to help identify defect-prone classes to guide quality assurance activities, such as tests or code reviews (Mende et Koschke, 2010). Indeed, limited resources and complex and large programs make difficult if not infeasible the test or review of the entire program. Prediction models are based on two main elements: the predictors and the prediction method.

Predictors or independent variables are variables that can be related to a dependent variable, such as defects. They constitute the inputs of prediction methods used to predict the dependent variable. Predictors used in defect prediction models include code complexity measures, complexity of code changes, object-oriented metrics, dependencies, process metrics, or organizational factors (Mende et Koschke, 2010).

A prediction method is a learning technique used to build a statistical model. Based on one or more predictors, this statistical model predicts or estimates a dependent variable (James *et al.*, 2014). The multitude of prediction methods include among others: decision trees, random forest, Bayesian networks, support vector machines, neural networks, and logistic regression.

2.4.1 Performance Measures

Various metrics have been proposed in the literature to assess the performance of prediction models (Arisholm *et al.*, 2007; Mende et Koschke, 2010). The following paragraphs describe the ones used in this dissertation.

Confusion Matrix: The confusion matrix, also called contingency table, reports how the model classified the different entities in defect categories compared to their actual classification (*i.e.*, predicted versus observed) (Bowes *et al.*, 2012). There are four categories:

– True Positive (TP): A faulty entity is predicted as faulty.
– False Positive (FP): A non-faulty entity is predicted as faulty.
– True Negative (TN): A non-faulty entity is predicted as non-faulty.
– False Negative (FN): A faulty entity is predicted as non-faulty.

Table 2.2 presents the structure of a confusion matrix.

Traditional Measures: The main traditional measures to assess the performance of a prediction model are precision, recall, and F-Measure.

Precision (P) indicates how reliable a prediction model is. It represents the proportion of predicted faulty entities that are actually faulty (Equation 2.1).

$$precision = \frac{TP}{TP + FP} \tag{2.1}$$

Recall (R) represents the proportion of entities actually faulty and predicted as such (Equation 2.2).

$$recall = \frac{TP}{TP + FN} \tag{2.2}$$

F-measure combines precision and recall and represents their weighted harmonic mean (Equation 2.3).

$$f - measure = 2 \times \frac{precision \times recall}{precision + recall} \tag{2.3}$$

2.5 Empirical Studies

Empirical studies can be defined as a category of scientific methods used to compare what we believe to what we observe. In the two last decades, there has been an increase interest and

use of empirical techniques in software engineering research. Empirical studies are used to compare techniques, evaluate new techniques, or to confirm or infirm perceptions and beliefs. Empirical studies help improve the validity and the generalizability of research results. In this dissertation, we use experimentations to assess the impact of antipatterns on OO unit testing (cf. Chapter 4) and to show the cost-effectiveness of the approach we propose for class integration test order problem (cf. Chapter 5). We follow in each of our experiments the guidelines provided in (Wohlin *et al.*, 2000) to plan and conduct a successful experiment. According to these guidelines, we use statistical tests to support our findings and identify and mitigate the threats that can impact the validity of our results.

2.5.1 Statistical Tests

Statistical tests are mathematical tools used to determine whether the differences observed when comparing two or more items are due to chance. They are important to strengthen studies conclusions. In the following, we describe the main statistical tests used in this dissertation.

Fisher's Exact Test: Fisher's exact test (Sheskin, 2007) is a non-parametric statistical test designed to determine whether two categorical variables are independent by comparing their proportions.

Mann-Whitney U Test: Mann-Whitney U test, also known as the two-sample Wilcoxon test, is a non-parametric test used as an alternative to the two-sample t-test when the data is not normally distributed. Given two samples, Mann-Whitney test assesses whether they come from the same distribution based on their ranks rather than the data.

Kruskal-Wallis Test: Kruskal-Wallis test is a non-parametric test for comparing multiple medians. It is the non-parametric test equivalent to the one-way Analysis of Variance (ANOVA) and an extension of the Mann-Whitney U Test.

Permutation Test: Permutation test (Baker, 1995) is a non-parametric statistical test alternative to the two-way ANOVA. The general idea behind such a test is that data distributions are built and compared by computing all possible values of the statistical test under rearrangements of the labels (representing the various factors being considered) of the data points.

Holm's Correction: Holm's correction (Holm, 1979) is used to adjust p-values in presence of multiple comparisons. The procedure sorts the p-values resulting from n tests in ascending

order of values. Then, to adjust the p-values, the first p-value (the smallest one) is multiplied by n, the next one by $n-1$, and so on. Any adjusted p-values less than α, the significance level, are significant (*i.e.*, that null hypothesis is rejected).

Odds Ratio: Odds ratio (OR) is defined as the ratio of the odds p of an event occurring in one group to the odds q to it occurring in another group. OR is computed as follows:

$$OR = \frac{p/(1-p)}{q/(1-q)} \tag{2.4}$$

$OR = 1$ indicates that the event is equally likely in both samples. $OR > 1$ indicates that the event is more likely to occur in the first group while an $OR < 1$ indicates the opposite (Sheskin, 2007).

Cliff's d: Cliff's d is a non-parametric effect size measure Cliff's d (Grissom et Kim, 2005) that indicates the magnitude of the effect size of the treatment on the dependent variable. Cliff's d is defined as:

$$d = \frac{\#(y_i > x_j) - \#(y_i < x_j)}{n_y.n_x} = \frac{\sum_i \sum_j d_{ij}}{n_y.n_x} \tag{2.5}$$

where $d_{ij} = sign(y_i - x_j)$ and $\#(y_i > x_j)$ is the number of how many times $y_i > x_j$ is true. The effect size is small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$ (Grissom et Kim, 2005).

2.5.2 Threats to Validity

The evaluation of the validity of an experiment aims at answering the question of how the conclusions of the experiment might be wrong. Because generally the validity of experiments is subject to threats, we should identify them and mitigate them as much as possible.

In the following, we describe four types of threats (Wohlin *et al.*, 2000) originally introduced by Cook et Campbell (1979).

Conclusion validity threats refer to the relation between the treatment and the outcome.

Internal validity threats concern any confounding factors that could have affect the dependent variables and thus the results of the experiment.

Construct validity threats are about the relation between theory and observation.

External validity threats concern the possibility of generalizing our results.

Yin (2008) defined another type of threats namely reliability threats. *Threats to reliability validity* refer to the ability to replicate a study with the same data and to obtain the same results.

Table 2.1 Description of Antipatterns Analyzed in this Dissertation.

Antipattern	Description
AntiSingleton (AS)	A class that provides mutable class variables, which, consequently, act as global variables.
BaseClassShouldBeAbstract (BCSBA)	A base class that is never instantiated
Blob (B)	A class that is too large and not cohesive enough. This class centralizes most of the processing, takes most of the decisions, and is associated to data classes.
ClassDataShouldBePrivate (CDSBP)	A class that exposes its fields, thus violating the principle of encapsulation.
ComplexClass (CC)	A class that has (at least) one large and complex method, in terms of cyclomatic complexity and LOCs.
LazyClass (LzC)	A class that has few fields and methods (with little complexity).
LongMethod (LM)	A class that has a method that is overly long, in term of LOCs.
LongParameterList (LPL)	A class that has (at least) one method with a long list of parameters with respect to the average number of parameters per methods in the program.
MessageChain (MC)	A class that uses a long chain of method invocations to implement (at least) one of its functionalities.
RefusedParentBequest (RPB)	A class that redefines inherited methods using empty bodies, thus breaking polymorphism.
SpaghettiCode (SC)	A class declaring long methods with no parameters and using global variables. These methods interact too much using complex decision algorithms. This class does not exploit and prevents the use of polymorphism and inheritance.
SpeculativeGenerality (SG)	A class that is defined as abstract but that has very few children. Her children do use its methods.
SwissArmyKnife (SAK)	A class whose methods can be divided in disjunct sets of many methods, *i.e.*, providing different, unrelated functionalities.

Table 2.2 Confusion Matrix.

	Actual True	Actual False
Predicted True	TP	FP
Predicted False	FN	TN

CHAPTER 3 LITERATURE REVIEW

The present chapter describes previous works related to ours. It presents the state-of-the art on antipatterns and their impact on software quality and the main approaches proposed to solve the class integration test order problem.

3.1 Antipatterns and their Impact on Software Quality

This section summarizes related literature concerning antipatterns (APs), their definition, their detection, and their impact on software quality.

3.1.1 Antipatterns: Definition and Detection

Many researchers and practitioners described APs and proposed solutions to refactor and–or remove them. Webster (1995) first discussed about APs in the context of OO programming in 1995. He described conceptual, implementation, and quality-assurance problems. (Riel, 1996) introduced 61 heuristics that allow developers to assess their programs manually and to improve object-oriented program design and implementation. Brown *et al.* (1998) introduced 40 APs, including Blob and Spaghetti Code. For each of the described APs, they listed the symptoms, the typical causes, and the consequences. They also proposed a solution that aims to refactor the code and remove the AP. Fowler (1999) suggested refactorings for 22 code smells. He described code smells in an informal way and provided a scenario to detect them manually. These descriptions have been the basis of further works concerning APs including their detection.

Researchers have proposed a number of approaches to specify and detect code smells and APs based on different techniques, ranging from manual inspection (Travassos *et al.*, 1999) to rule-based systems (Marinescu, 2004; Moha *et al.*, 2010).

Travassos *et al.* (1999) introduced manual inspections and reading techniques to identify code smells.

Marinescu (2004) presented a metric-based approach and a set of detection strategies to identify and detect 10 types of antipatterns. The proposed approach captures deviations from good design principles characterized by absolute and relative thresholds of different metrics.

Moha *et al.* (2010) presented DECOR (Defect dEtection for CORrection), an approach based

on rules cards that specify each antipattern. DECOR offers also a platform to automatically convert each rule card into a detection algorithm. Moha *et al.* (2010) reported that DECOR can achieve a recall of 100% and a precision between 41.1% and 87% for the following antipatterns: Blob, Spaghetti Code, and Swiss Army Knife.

Researchers also investigated the use of machine learning techniques—see for example Khomh *et al.* (2011) and Maiga *et al.* (2012) — to locate code smells and APs. Khomh *et al.* (2011) presented an approach, BDTEX, based on Bayesian belief to detect APs networks and Maiga *et al.* (2012) used support vector machines (SVM) to detect APs.

More recently, Palomba *et al.* (2015) proposed HIST, an approach based on change history information, to detect occurrences of five kinds of APs: DivergentChange, ShotgunSurgery, ParallelInheritance, Blob, and FeatureEnvy.

In the context of our studies, we analyze the impact of some of the APs described in these works on the cost of object-oriented unit testing. We also make use of DECOR, one of the state-of-the-art tools, to detect classes involving APs.

3.1.2 Impact of Antipatterns on Software Quality

The assessment of the impact of antipatterns on software quality attributes has been the subject of many previous works. Most of the results suggest that code smells and APs negatively impact software quality confirming somehow the common lore about them.

Deligiannis *et al.* (2004) conducted an empirical study to analyze the influence of God classes on software understandability and maintainability. The findings of this experiment support the claim that God classes have a negative impact on the evolution of design structures.

Du Bois *et al.* (2006) performed a controlled experiment with graduate students on the impact of decomposition of Blobs (God classes) on understandability. Their finding show that the decomposition of Blobs into a number of collaborating classes using well-known refactorings can improve program comprehension.

Abbes *et al.* (2011) investigated the influence of Blob and Spaghetti Code on software understandability. Their results show that the presence of Blob or Spaghetti Code does not negatively impact program understandability, but the combination of the two APs decrease significantly this quality attribute. Yamashita et Moonen (2013) obtained similar results. Indeed, the results of their empirical study show that the co-occurrence of antipatterns in a same class impedes its maintenability.

(Olbrich *et al.*, 2009) analysed the historical data of Lucene and Xerces over 6 years. They discovered that classes involving Blob and Shotgun Surgery change more often than other

classes. Khomh *et al.* (2012) also analyzed the change- and defect-proneness of APs in four open source programs (ArgoUML, Eclipse, Mylyn, and Rhino). Their results reveal that classes participating in APs are more change- and defect-prone than other classes. The results indicate also that the size alone cannot explain the participation of classes to APs.

Chatzigeorgiou et Manakos (2010) studied the evolution of Long Method, Feature Envy, and State Checking in 10 versions of JFlex and 14 versions of JFreeChart. They found that a significant percentage of these APs are introduced by adding new methods to the programs (Chatzigeorgiou et Manakos, 2010).

These studies provide evidence of the negative impact of APs on many software quality attributes. However, to the best of our knowledge, no previous work investigated the impact of APs on class testability and testing cost. We share with these studies the goal of analyzing the impact of antipatterns on different attributes of software quality. Our goal is to analyze the impact of APs on OO unit testing cost.

3.1.3 Antipatterns Refactoring

Because of the negative impact of APs on software quality attributes, researchers and practitioners advise to refactor the code to remove APs (Brown *et al.*, 1998; Khomh *et al.*, 2012). Program refactoring is a technique used to improve the internal structure of a program without changing its external behaviour (Fowler, 1999). Program refactoring aims at increasing program quality and hence reduce the time and costs of program maintenance and evolution (Fowler, 1999). Researchers proposed various refactoring techniques and describe the circumstances of their applicability (Riel, 1996; Brown *et al.*, 1998).

Du Bois *et al.* (2006) showed that applying refactoring techniques on a Blob can actually improve understandability. Tsantalis et Chatzigeorgiou (2009) proposed a semi-automatic approach for move-method refactoring with the aim of removing Feature Envy smells, while Fokaefs *et al.* (2011) proposed JDeodorant, a tool for extract-class refactoring in presence of God Classes.

In this dissertation, we suggest refactoring actions to reduce Madum testing cost. The proposed refactoring are different from the ones proposed in the works described above and can complement them to improve the overall quality of software.

3.2 Class Integration Test Order

One of the main challenges in class integration test of any large OO program is to determine the order in which classes should be tested. This order is critical since it impacts the cost of the integration process and the order in which defects will be detected (Tai et Daniels, 1997). Harrold *et al.* (1992) were the first to propose a class test order based on class inheritance in which base class should be tested before a derived class. Kung *et al.* (1995), pioneer in solving the class integration test order (CITO) problem, extended this precedence rule by taking into account aggregation and association relations. Thus, when a class C depends on a class B through a relation of inheritance, association, or aggregation, B should be tested before C. In the work of Briand *et al.* (2003b), C and B are named client and server respectively. In the present thesis, this rule is named server before client (SBC) principle.

Kung *et al.* (1995) presented an approach in the context of regression testing that computes an order in which modified and impacted classes should be (re-)integrated and tested with minimum cost. They introduced the concept of class firewall to find all affected classes after changes in the program.

Following Kung *et al.* (1995), many researchers studied the problem of class integration test order but in various developments contexts. The goal of the proposed approaches is to devise an integration test order which minimizes SBC violations, and therefore the number and–or complexity of stubs (SBC violations). When there is no dependency cycle between classes, a simple reverse topological sort produces the optimal class integration test order (Kung *et al.*, 1995). However, in most of real object-oriented programs, there are dependency cycles (Briand *et al.*, 2002b). Testing classes involved in a dependency cycle requires to break the cycle by removing one dependency and writing a stub to replace the target class of that dependency. Stubs are also required when one needs to test a class that depends on classes not already tested or unavailable. Since writing stubs is expensive and defect-prone (Briand *et al.*, 2001), the goal of the approaches proposed in literature is to find an order in which classes can be integrated and tested with less stubs in presence of dependency cycles or unavailable classes. CITO-solving approaches in presence of cycles can be divided into two main categories: graph-based approaches and search-based approaches.

In the graph-based approaches, the program under test is represented by an object relation diagrams (ORD) or a variant of ORD. The approaches in this category provide heuristics to break dependency cycles to make the graph acyclic and apply a topological sort to derive an integration test order. This category includes among other the approaches of Kung *et al.* (1995), Tai et Daniels (1997), Traon *et al.* (2000), Hanh *et al.* (2001), Briand *et al.* (2001),

Malloy *et al.* (2003), Mao et Lu (2005), and Abdurazik et Offutt (2009).

Differently from the graph-based approaches, the search-based approaches use search-based algorithms to find orders near to the optimal that can minimize stubs cost. These approaches benefit from the use of optimization techniques to overcome the limitation inherent to the graph-based approaches: difficulty to use multiple or complex factors to estimate the cost of stubs, solutions sometimes sub-optimal (Briand *et al.*, 2002b). Hanh *et al.* (2001) and Briand *et al.* (2002b) proposed the use of genetic algorithm to solve the CITO problem. Borner et Paech (2009a) proposed a new approach to class integration test order that takes into account defect-prone classes. The approach divides classes into two sets (i) the test focus set, *i.e.*, classes that must be tested first, and (ii) all other classes. Recently, researchers proposed multi-objectives optimization techniques to solve the CITO problem (Assunção *et al.*, 2011, 2014; da Veiga Cabral *et al.*, 2010; Vergilio *et al.*, 2012), since the cost of stubs can be composed of different elements such as attribute coupling and method coupling (Briand *et al.*, 2002b). da Veiga Cabral *et al.* (2010) first proposed the use of multi-objectives techniques to solve the CITO problem. They implemented a Pareto ant colony (PACO) algorithm with the goal of minimizing two objectives: the attribute and method couplings defined by Briand *et al.* (2002b). The experiment results showed that the PACO algorithm found more non-dominated solutions than the GA, thus giving a large variety of solutions to the tester. In a recent comparative study on solving the CITO problem using multi-objectives techniques, Assunção *et al.* (2014) showed that the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) is the most suitable algorithm in general, whereas the Pareto Archived Evolution Strategy (PAES) is the best regarding complex systems.

A classification of the existing approaches based on the different cost objectives can be found in (Wang *et al.*, 2011).

Overall, in the best of our knowledge, except the approach proposed by Borner et Paech (2009a), all approaches proposed to solve the CITO problem aim at minimizing the number or complexity of stubs only. However the boolean classification used in (Borner et Paech, 2009a) impairs to create balanced orders able to promote early defect detection and minimize SBC violations. We inspire from these works to build our approach to the CITO problem that seeks to minimize stubs cost, *i.e.*, SBC violations, and increase early defect detection by prioritizing defect-prone classes, such as AP classes.

CHAPTER 4 A STUDY ON THE RELATION BETWEEN ANTIPATTERNS AND THE COST OF CLASS TESTING

4.1 Introduction

Antipatterns (APs) are defined as recurring and poor design choices (Brown *et al.*, 1998). As shown in the previous chapter, there is a large number of recent and past studies that show the negative impact of antipatterns on software quality, in particular software comprehensibility, understandability, and maintenability. These studies reveal also the low resilience to change and defect of classes involving antipatterns. Despite the rich literature on the undesired side effects of APs on software quality attributes (Du Bois *et al.*, 2006; Abbes *et al.*, 2011; Khomh *et al.*, 2012), to the best of our knowledge, there is no study aimed at answering the following question:

What is the impact of APs on OO unit testing cost?

The aim of our first study in this dissertation is consequently to study and gather evidence on the impact of APs on OO unit testing. We conjecture that, on the one hand, AP classes are more difficult to test and that, on the other hand, they must be tested thoroughly because they are more difficult to understand and maintain and also more defect-prone (Du Bois *et al.*, 2006; Abbes *et al.*, 2011; Khomh *et al.*, 2012).

Based on our conjectures, we investigate (i) whether the effort to perform class unit testing is higher for classes participating in APs than in other classes and (ii) the cost-benefit achieved when prioritizing testing of classes participating in APs compared to other classes. Following Bache et Mullerburg (1990), we use as surrogate measure of testing cost the number of test cases required to test a class.

There are many methods and adequacy criteria for class unit testing (Binder, 1999; Bashir et Goel, 2000). Some of them, although powerful and easy to implement, require the availability of appropriate documentation, such as state machines or dynamic diagrams (Binder, 1999). This documentation being often missing, we rather use a testing strategy—named minimal data members usage matrix, Madum testing (Bashir et Goel, 2000)—and described in Section 2.1.3. As mentioned in that section, Madum testing does not require sophisticated design documentation to exist; it is based on the source code.

We compute the number of Madum test cases for classes of four Java open-source programs, Ant 1.8.3, ArgoUML 0.20, CheckStyle 4.0, and JFreeChart 1.0.13. Then, we detect, using the tool DECOR (Moha *et al.*, 2010) (cf. Section 2.2), APs occurrences in these programs.

To verify our first conjecture, we compared the number of Madum test cases required to test AP classes to that of non-AP classes. To support our second conjecture, we assessed the defect-proneness of AP classes and the benefit of prioritizing their test over non-AP classes.

Results of our study bring evidence that the unit testing of classes participating in APs—in particular Blob, AntiSingleton and ComplexClass—require a higher number of test cases than that of other classes. Results also show that AP classes are more defect-prone than other classes, confirming the outcome of the experiment in (Khomh *et al.*, 2012); thus they should be carefully tested. Using a naïve testing order, we also show that prioritizing the test of classes involving APs is cost-effective.

The rest of the chapter presents the design and the results of our empirical study.

4.2 Study Design

The *goal* of this study is to investigate the cost of unit testing for classes participating in APs, as opposed to other classes, and the potential benefits obtained when prioritizing the test of AP classes. The *quality focus* is the effort needed to produce test cases, and the extent to which testing particular classes would help to reveal defects. The *perspective* is of researchers, interested to understand the influence of APs on software quality from the point of view of testing and to conduct more research in this direction. The results will also be useful for practitioners to decide whether they should apply refactoring actions to classes participating in APs, to improve software quality, *e.g.*, to reduce change- and defect-proneness (Khomh *et al.*, 2009, 2012) and reduce the testing effort.

The *context* of this study consists of a class unit testing strategy—Madum testing—and one release of four Java open-source programs. We have chosen Madum testing because it requires information always available from the source code while other OO testing strategies—such as those based on pre-and-post conditions (Boyapati *et al.*, 2002) as well as state-based strategies (Binder, 1999)—rely on documentation rarely available in practice.

We choose the four open-source programs according to different criteria: (i) programs belonging to different application domains, (ii) availability of bug-fixing data from versioning and issue-tracking systems, and (iii) use in previous studies concerning APs and—-or testability (Abbes *et al.*, 2011; Bruntink et van Deursen, 2006). Table 4.1 reports the number of classes that participate in APs or not for the four programs [1]. *Apache Ant* [2] is a built tool

1. The sum of classes participating in different kinds of APs can be greater than the number of classes participating in at least one AP, because some classes participate in more than one AP.

2. http://ant.apache.org/

Table 4.1 Number of classes participating in different kinds of APs.

Name (Abbr)	Ant	ArgoUML	CheckStyle	JFreeChart
AntiSingleton (AS)	2	257	15	20
BaseClassShouldBeAbstract (BCSBA)	20	20	3	14
Blob (B)	50	123	11	32
ClassDataShouldBePrivate (CDSBP)	84	44	4	20
ComplexClass (CC)	103	214	34	55
LazyClass (LzC(	46	60	4	21
LongMethod (LM)	178	267	69	102
LongParameterList (LPL)	34	237	8	50
MessageChains (MC)	186	145	7	56
RefusedParentBequest (RPB)	67	497	80	62
SpaghettiCode (SC)	1	45	1	0
SpeculativeGenerality (SG)	4	23	1	1
SwissArmyKnife (SAK)	1	4	0	14
Antipattern classes	452	901	161	245
No Antipattern (None)	297	376	99	233

for Java. Its release 1.8.3 has 209 KLOC for 767 classes. *ArgoUML*[3] is an open-source tool for UML diagrams. We used its release 0.20, which consists of 1,277 classes for 196 KLOC. *CheckStyle*[4] is a development tool for Java programs. It checks whether Java code adheres to a specific coding standard chosen by the developers. Its release 4.0 has 261 classes for 56 KLOC. *JFreeChart*[5] is a Java class library to embed/generate charts in Java programs. Its release 1.0.13 consists of 484 classes for 183 KLOC.

4.2.1 Research Questions

This study aims at addressing the general research question:

What is the impact of APs on OO unit testing cost?

We refine this research question in the following three specific research questions:

RQ1: *How large is the Madum test suite for classes participating in APs compared to that of other classes?* This research question investigates whether classes participating in APs have larger Madum test suites than other classes. Our intuition is that poor design and coding practices have consequences on testing and thus can require a high number of test cases to fulfill Madum testing, because APs make it more difficult to partition methods into data

3. http://argouml.tigris.org/
4. http://checkstyle.sourceforge.net/
5. http://www.jfree.org/jfreechart/

slices, *i.e.*, most of the methods belong to all slices.

RQ2: *How does the size of the Madum test suite vary among classes participating in different kinds of APs?* This research question refines **RQ1**. Some APs can have a higher impact on testing cost than others.

RQ3: *What is the potential cost-benefit achieved when focusing testing on APs, as opposed to other classes?* Given the cost needed to test a class, we investigate what would be the benefit, in terms of discovered defects, assuming that the Madum test suites can detect all defects of the class under test.

4.2.2 Variables and Analysis Method

For **RQ1**, the *dependent variable* is the number of test cases required to test each class using Madum testing. The *independent variable* is a boolean variable that indicates the participation of classes in some APs. Such a boolean variable is true if a class participates to at least one AP; false otherwise.

We statistically compare the number of test cases between AP and non-AP classes. Specifically, we test the following hypotheses H_{01}:

There is no significant difference between the number of test cases of classes participating and not in APs.

We test the hypothesis using a non-parametric test, the Mann-Whitney U test. Because we do not know a priori whether the number of test cases will be higher for AP classes or non-AP classes, we perform a two-tailed test. Besides testing the hypothesis, we also estimate the magnitude of the differences of means between classes participating and not in APs using the non-parametric effect size measure Cliff's d (Grissom et Kim, 2005).

For **RQ2**, the *dependent variable* is also the number of Madum test cases. The *independent variable* is a boolean variable for each kind of AP, indicating whether a class participates in that kind of AP or not. We test the following null hypothesis:

H_{02}: *There is no significant difference between the number of test cases of classes participating in different kinds of APs.*

First, we test the hypothesis using the Kruskal-Wallis test. Then, we pairwise compare the number of test cases for different kinds of APs using the Mann-Whitney test. Finally, we correct the obtained p-values using the Holm's correction (Holm, 1979).

For **RQ3**, we identify the number of post-release defects affecting each class for the analyzed releases of the four programs. Then, we take the perspective of a "lazy" tester, who starts to

test the classes in an increasing order of number of test cases regardless of other more sound criteria, for example, the number of needed stubs or the different kinds of interactions with other classes. We thus assume a simple and blind testing strategy as our goal is to verify the potential increase in detected defects rather than using more sophisticated approaches, such as the firewall strategy (Kung *et al.*, 1995; Binder, 1999) or other strategies to determine the integration testing order in OO programs (Briand *et al.*, 2003a; Hanh *et al.*, 2001). We analyze the potential benefit achieved in terms of defects that can be discovered when the number of test cases increases. We perform such an analysis by plotting cumulative curves of number of test cases (*independent variable)* vs. number of defects that can be discovered if properly testing these classes (*dependent variable*) for classes that participate in APs or not.

All statistics have been performed using the *R* statistical environment [6]. For all statistical tests, we assume a significance level of 5%.

4.2.3 Data Collection

To answer our RQs, we implement the algorithm to identify test cases required by Madum testing and count the number of test cases as detailed in Section 2.1.3. This number is computed from the Madum of the class. We build for each class its Madum using a static analysis of the source code that allows identifying the slices of the class.

For what concerns the detection of APs, as in previous works (Abbes *et al.*, 2011; Khomh *et al.*, 2012), we use DECOR described in Section 2.2.

Finally, we compute the number of post-release defects as follows:

1. We identify, from the commit notes of the versioning system of the analyzed programs, changes related to bug-fixing, by matching issue tracking system IDs and keywords, such as “bug” and “fixed” (Fischer *et al.*, 2003). We limit our attention to the time frame between the release date and the next release.
2. We check, by analyzing the information in the issue tracking system, whether the fix concerns a corrective maintenance (defect fixing) or whether it is an implementation feature-request/enhancement. Then, we discard the latter. Also, we restrict our attention to issues marked as “CLOSED” and “FIXED” in the issue tracking system.
3. Finally, for the remaining issues, we map the changes to the affected classes by analyzing the changes that occurred. We count the number of defect-fixing changes that occurred to each class.

6. http://www.r-project.org

The number of defect-prone classes is, for Ant, 131 out of 767 (17%), and for ArgoUML, 190 out of 1284 (15%). CheckStyle and JFreeChart contain less defects with respectively, three defect classes out of 261 (1%), and 16 defect classes out of 484 (3%).

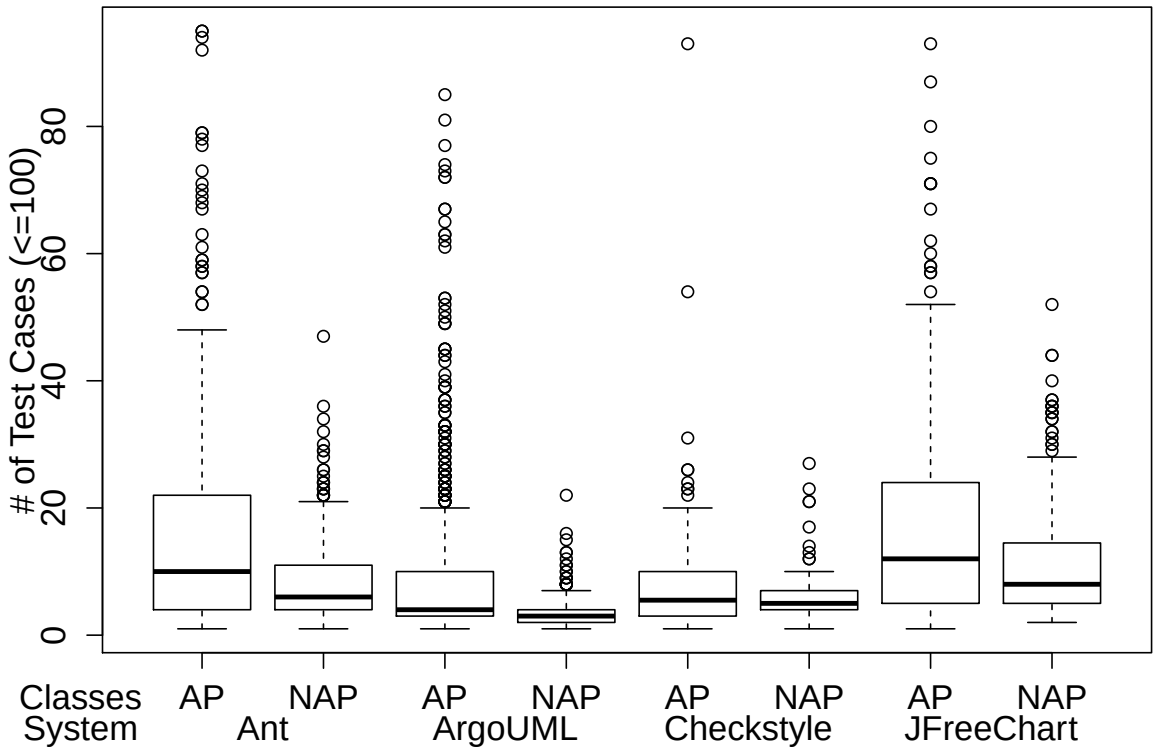


Figure 4.1 Number of test cases in the Madum test suite for classes participating (AP) and not (NAP) in antipatterns.

4.3 Study Results

4.3.1 RQ1: How large is the Madum test suite for classes participating in APs compared to that of other classes?

Fig. 4.1 shows the distribution of Madum test suite sizes for classes participating and not in APs for the four analyzed programs. The figure clearly highlights how, with some exceptions (CheckStyle in particular), the test suite sizes are larger for classes participating in APs than for other classes.

Table 4.2 Madum test suite size for classes participating and not to APs: Mann-Whitney tests and Cliff's d results.

program	Mean TCs AP	Mean TCs NAP	p-Value	Cliff's d
Ant	18	9	$<$ **0.01**	0.23 (Small)
ArgoUML	10	3	$<$ **0.01**	0.35 (Medium)
CheckStyle	9	6	$=$ **0.05**	NA
JFreeChart	26	13	$<$ **0.01**	0.22 (Small)

Table 4.2 reports the Mann-Whitney test results and Cliff's d effect size obtained when comparing the number of Madum Test cases between classes that participate in APs or not. Except for CheckStyle, results show statistically-significant differences. The Cliff's d effect size is small for Ant and JFreeChart, and medium for ArgoUML.

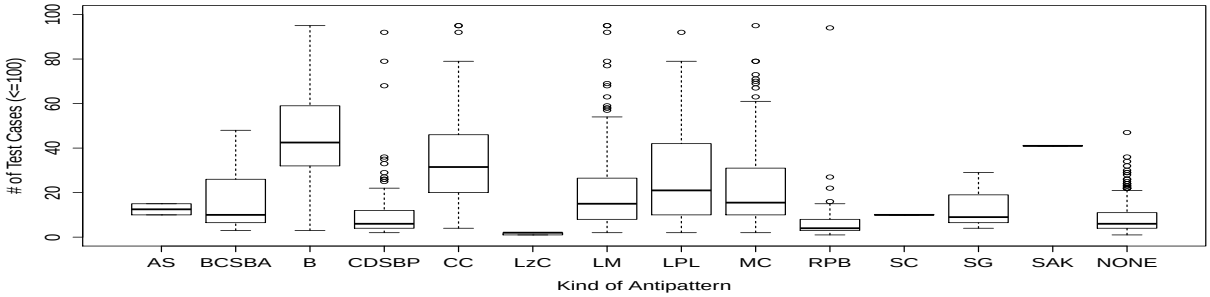
RQ1 Summary: We can reject the null hypothesis H_{01} for Ant, ArgoUML, and JFreeChart. In these three programs, the number of test cases required for Madum testing of AP classes are significantly higher than those of other classes. For CheckStyle, the difference is not statistically significant, therefore we cannot reject H_{01} for this program. We can conclude that AP classes are less testable than non-AP classes. If developers want to test AP classes thoroughly using Madum testing, they must write more test cases leading to a higher testing cost.

4.3.2 RQ2: How does the size of the Madum test suite vary among classes participating in different kinds of APs?

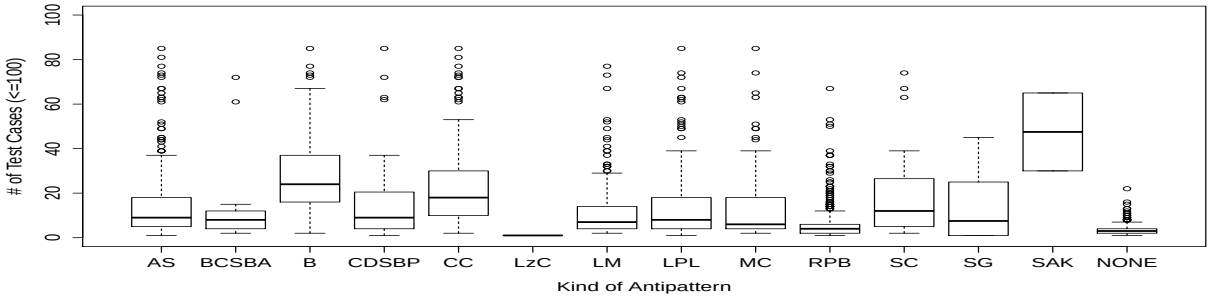
Fig. 4.2 shows boxplots of the number of test cases needed to test classes participating in different kinds of APs.

For Ant, classes participating in Blob (B) require a significantly higher number of test cases than classes participating in other APs (BCSBA, CSBP, LzC, LM, MC, RPB) and than classes not participating in APs. A high number of test cases is also required for ComplexClass (CC), and, specifically, significantly higher than CDSBP (ClassDataShouldBePrivate), LzC (LazyClass), LM (LongMethod), MC (MessageChains), RPB (RefusedParentBequest), and classes not participating in APs. In all cases, the Cliff's delta effect size is high. Some APs do not imply a high number of test cases: LzC or RPB.

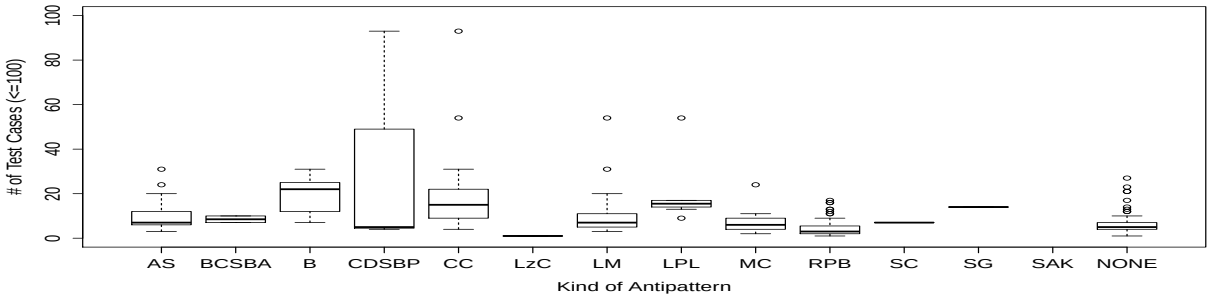
We obtain similar results for ArgoUML, where Blob (B) require a significantly higher number



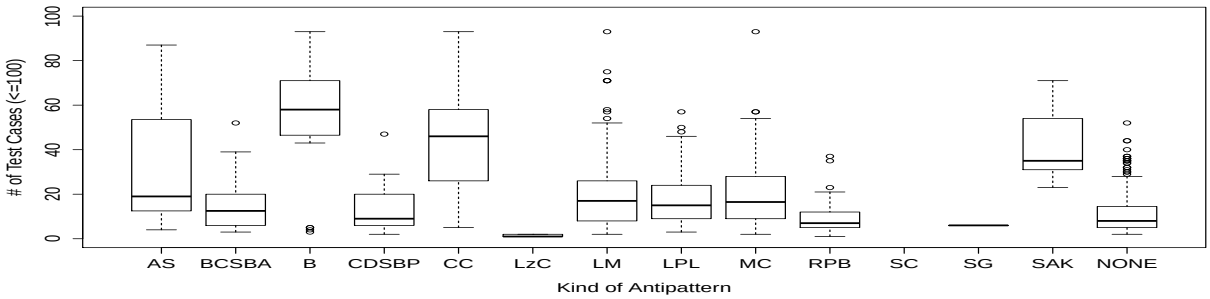
(a) Ant



(b) ArgoUML



(c) CheckStyle



(d) JFreeChart

Figure 4.2 Number of test cases in the Madum test suite for classes participating in different kinds of APs.

of test cases than LzC, LM, LPL, MC, RPB, BCSBA (high effect size), than CDSBP, SG, SC (medium effect size), and classes not participating in APs (high effect size). Complex Classes (CC) require a significantly higher number of test cases than (1) LzC, RPB, BCSBA (Base Class Should Be Abstract) (high effect size), than (2) CSBLP, LM, LPL, MC (medium effect size), and than (3) classes not participating in APs (high effect size). Also, Anti-Singleton (AS) require more test cases than LzC, RPB, and no AP classes (high effect size in all cases), and CDSBP require more test cases than LzC, RPB and no AP classes (high effect size). We found no statistically significant differences for SwissArmyKnife (SAK) despite what the boxplot shows because of the limited number of occurrences of this AP in the analyzed release.

For CheckStyle, due to the limited number of AP occurrences, the only significant difference found is for Anti-Singleton (AS) classes that require a significantly higher number of test cases than LzC and RPB, with high effect size.

A similar situation occurs for JFreeChart, in which AS classes require a significantly higher (with a high effect size) number of test cases than LzC and RPB.

Overall, the obtained results tell that different kinds of APs exhibit different testing cost. Intrinsically, some APs are classes with more responsibility—see for example Blob, Complex-Class, or AntiSingleton. The presence of these APs is also related with a higher number of test cases. Other kinds of APs, such as LazyClass, RefusedParentBequest or Method Chains are not really related to classes having too much responsibility, hence the presence of APs does not imply having more test cases.

RQ2 Summary: Classes participating in APs, such as Blob, ComplexClass, AntiSingleton, or SwissArmyKnife, require a significantly higher number of test cases than other classes. On the contrary, APs, such as LazyClasses, MethodChains, or RefusedParentBequest require a relatively small number of test cases.

4.3.3 RQ3: What is the potential cost-benefit achieved when focusing testing on APs, as opposed to other classes?

The previous research questions have shown that, in general, classes participating in APs are more expensive to test than other classes. Some specific kinds of APs are particularly more expensive than others. We now investigate to what extent such additional cost makes

worthwhile the test of APs. In other words, if the test of AP classes (more expensive to test than other classes) also means potentially discovering more defects. In that case, such a cost maybe cost-effective.

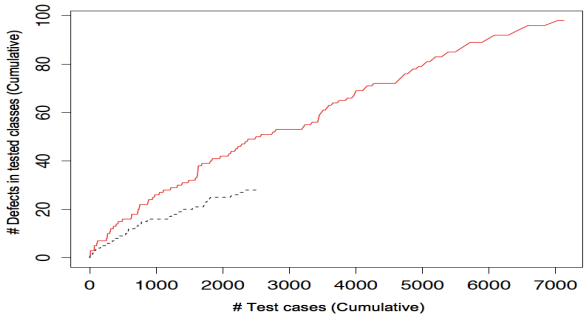
Table 4.3 reports results of the Fisher's exact test, which statistically compares the proportion of defect-prone classes between classes participating and not in APs and the corresponding values of the odd ratio (replicating Khomh *et al.* (2012)). For Ant and ArgoUML, results are statistically significant and classes participating in APs have 2.45 and 4.31 more time the chance of exhibiting at least a post-release defect than other classes. As for CheckStyle and JFreeChart, only AP classes exhibited post-release defects, therefore the Fisher's exact test OR is infinite. In summary classes participating in APs have higher chances of exhibiting post-release defects than other classes in agreement with the findings of Khomh *et al.* (2012).

Fig. 4.3 shows what would be the cumulative number of defects found when testing only classes participating in APs (red line) or only classes not participating in APs (black line). We adopt a "lazy" ordering, testing classes in ascending order according to their number of test cases, *i.e.*, testing those with a lower number of test cases first. When interpreting such results, it is important to point out that: (i) we are considering class unit testing therefore we are not following any strategy for integration ordering (the next chapter will investigate how such ordering would impact on testing cost-effectiveness) and (ii) we consider that, when we test a class, all defects are discovered. We are aware this is not always true; however we show is basically an upper bound of the number of defects discovered if testing such classes.

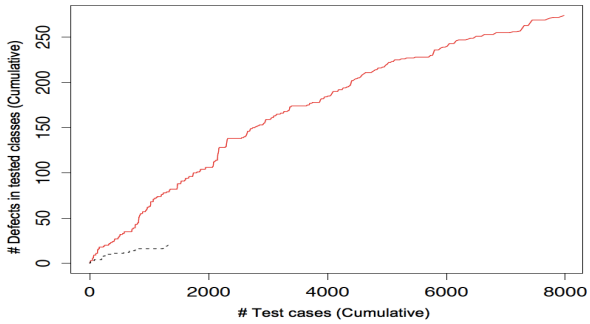
When considering the graphs representing the results of Ant and ArgoUML in Figure 4.3, even when starting to test the first few classes, the number of defects that can be found for classes participating in AP is substantially higher than that for other classes. In the graphs of results for CheckStyle and JFreechart (cf. Figure 4.3), the non-AP line is flat because there is no defects in these classes in these two programs.

Table 4.3 Results of Fisher's exact test for defect-proneness of classes participating and not in APs. (DP: Defect-prone).

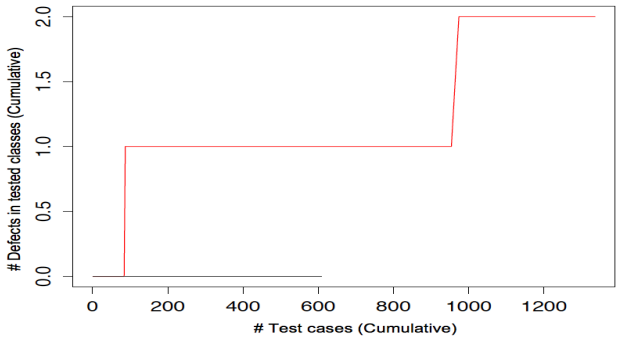
program	AP classes		NAP classes		p-value	OR
	DP	Not DP	DP	Not DP		
Ant	91	354	28	268	$<$ **0.001**	2.45
ArgoUML	167	726	19	357	$<$ **0.001**	4.31
CheckStyle	2	158	0	99	0.5259	Inf
JFreeChart	12	221	0	232	$<$ **0.001**	Inf



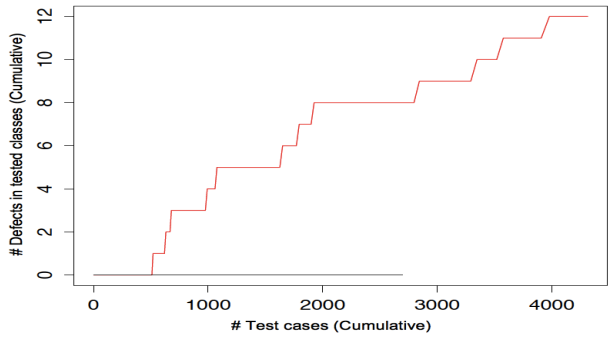
(a) Ant



(b) ArgoUML



(c) CheckStyle



(d) JFreeChart

Figure 4.3 Testing cost-effectiveness for classes involving (red) and not (black/dashed) APs.

RQ3 Summary: Classes participating in APs exhibit a significantly higher defect-proneness than other classes. Despite the fact that their testing cost is higher, it is cost-effective to analyze/test them with a higher priority than other classes.

In summary, this study has highlighted that APs increase the class unit test cost, although at the same time such classes have to be properly tested because they contain a high proportion of defects.

4.4 Threats to Validity

In this section, we discuss the main threats to validity that could affect our study.

Threats to *construct validity* of our study are mainly due to possible mistakes/imprecisions in the AP detection, in the classification of defects used to address **RQ3**, and in the "lazy" testing ordering considered in **RQ3**. Concerning AP detection, we relied on DECOR (Moha *et al.*, 2010). Although, DECOR is known to be accurate (Moha *et al.*, 2010), there is no guarantee that we detect all APs or that what we classified as APs are indeed true APs. We mitigate this threat by making the list of detected APs available to other researchers.

A second threat to construct validity derives from the definition of defect, the content of bug tracking systems and the way in which defects are assigned to classes. It is well-known that issue tracking systems contain all sort of change requests (Antoniol *et al.*, 2008). Thus, in general, we cannot guarantee that all issue tracking entries are indeed related to fixing defects. For two programs (Ant and ArgoUML), the issue tracking system uses a specific category to classify corrective maintenance changes ("DEFECT"). For the two others (CheckStyle and JFreeChart), we relied on information about fixed bugs available in the release notes. Last but not least, although we used a widely-adopted approach to identify bug-fixing changes and link them to issue reports (Fischer *et al.*, 2003), such approach can miss some fixes not explicitly mentioning the issue ID in the commit note (Bachmann *et al.*, 2010).

Finally, the simplistic "lazy" integration strategy, has to be considered as a way of gauging the maximum theoretical difference, if any, between defect detection obtained by prioritizing AP classes testing over classes not participating in APs. We cannot claim that Madum test cases will eventually detect all defects or what percentage of undetected defects exists. Also, such a simplistic testing approach does not account for class interactions, nor it considers the actual complexity of writing and running test cases. Class interaction affects the number of needed stubs, and thus the programmer coding effort; writing and running the actual test

cases impact coding as well as program understanding effort. In essence, we can only claim that, no matter what the testing strategy or the testing order, developers should focus their quality assurance efforts on classes participating in APs, and possibly remove APs as a first step toward an improved design.

To mitigate the threats to *conclusion validity* to our study, we used proper tests and effect size measures to address our research questions. In particular, we used non-parametric tests (Mann-Whitney test and Fisher's exact test) and effect size measures (Cliff's delta and Odds Ratio) that do not make any assumption about the underlying distributions of data. Finally, we dealt with problems related to performing multiple Mann-Whitney tests using the Holm's correction procedure.

Regarding threats to *external validity*, although we analysed programs belonging to different application domains and developed by different teams, the study needs to be replicated on further programs to confirm or contradict our results.

4.5 Conclusion

This chapter investigated the effects of APs on the cost of unit testing in OO programs. We detected APs using the tool DECOR (Moha *et al.*, 2010) in four Java programs, namely Ant 1.8.3, ArgoUML 0.20, Checkstyle 4.0, and JFreeChart 1.0.13. Then, we estimated the number of test cases required to test each class using Madum testing (Bashir et Goel, 2000). Finally, we compared the number of test cases required for classes that participate in APs or not and related such a number with the number of post-release fixed defects of such classes.

Findings of the study strongly support the evidence that:

1. Classes participating in APs are, in general, more expensive to test than other classes.
2. Specifically, classes participating in some kinds of APs, such as Blob, AntiSingleton or ComplexClass have a significantly higher testing cost than other classes, whereas some other APs, such as MethodChain or LazyClass, do not strongly contribute to the cost of testing.
3. Giving a higher priority to classes involving APs makes testing activities more cost-effective by allowing to test classes with the highest proportion of defects first.

Although APs are expensive to test, prioritizing their test is cost-effective as shown in the present chapter: prioritizing the test of AP classes allows detecting most of the defects and early. However, the order used in this experiment is not realistic as it does not account other cost related to class integration test order. The next chapter proposes a new approach to

the class integration test order problem that includes the outcome of the present study. This approach prioritizes defect-prone classes, such as AP classes, while minimizing cost related to class integration test order.

CHAPTER 5 MITER: MINIMIZING INTEGRATION TESTING EFFORT

5.1 Introduction

One key factor to be considered during integration testing is to determine the most suitable test order. Indeed, the order in which classes are tested impacts the cost of integration testing generally expressed in terms of cost of stubs to develop and the order in which defects are discovered (Tai et Daniels, 1997).

Kung *et al.* (1995) defined a test order strategy according to which a server class (independent class) should be tested before its client classes (dependent classes). They expressed the idea as follows: if we can test classes using tested classes by following a proper test sequence, then the test effort for constructing test stubs and test drivers can be reduced. If a tester can test a server class before its client classes, then a stub for the server class for testing the client classes is not needed. Thus, testing server classes before their client classes is the principle of Kung's strategy, underpinning also most of the approaches that followed (Tai et Daniels, 1997; Traon *et al.*, 2000; Briand *et al.*, 2002b; Abdurazik et Offutt, 2009). The main purpose of all these approaches is to reduce testing cost when testing modified programs or performing class integration testing.

The experimental results of the previous chapter showed that while AP classes are more expensive to test than other classes, prioritizing their testing allows detecting most of the defects and early. However, the order used to perform this experiment is a "lazy" and naïve order based on the ascendant ordering of the required number of test cases. The cost related to this test order can be very high because it does not follow the SBC principle. Unfortunately, while the issue of class integration test order (CITO) has been extensively investigated, to the best of our knowledge, only the approach proposed by Borner et Paech (2009a,b) seeks for testing effectiveness in terms of defect detection.

We propose in the present chapter a new approach to the problem of class integration test order that extends the need of prioritizing AP classes to all defect-prone classes while fulfilling the SBC principle. Named MITER (Minimizing Integration Testing EffoRt), our approach enforces the SBC principle and is built on top of existing testing strategies and file-level defect location approaches. MITER aims at determining class integration test orders with the goals of (1) maximizing early defect detection *i.e.*, test with high priority classes having a high (estimated) defect-proneness, such as AP classes and (2) minimizing the cost of SBC violations (stubs). Differently from the test focus approach (Borner et Paech, 2009a,b),

MITER uses a fine-grained prioritization instead of a Boolean classification of classes to test (classes that must be tested first and (ii) all other classes). MITER's underlying assumption is that it is more realistic to avoid such a dichotomous distinction between classes and, instead, assign a level of priority ranging from zero to one to classes to test. Following the call-coupling coverage criterion proposed by Lin et Offutt (1998) for class integration testing, MITER uses the number of call sites of classes to be tested as surrogate to measure integration testing cost. Early defect detection is then measured with respect to that cost.

To compute a close to optimal or optimal testing order, MITER relies on a memetic algorithm (MA), a meta-heuristic approach derived from genetic algorithm (GA), which has been proved to be efficient in solving many search problem instances. MITER uses as fitness function a cost function consisting in a linear combination of the normalization of two terms: a term accounting for priorities (*i.e.*, likelihood that a class is faulty) and a term quantifying the number of SBC violations. The first term allows increasing early defect detection capability. The higher the priority weight, the higher the risk of SBC violations while the lower the priority weight, the lower the probability of observing a substantial early defect detection.

To verify the ability of MITER to promote early defect detection while minimizing SBC violations, we perform an empirical evaluation using multiple releases of three Java open-source programs: Ant, ArgoUML, and Xerces. Our evaluation answers the following general question:

Can MITER provide balanced class integration orders that minimize server-before-client violations and maximize early defect detection capability?

We analyse MITER's orders in terms of early defect detection and SBC violations for different values of the priority weight. We also compare MITER to the test focus approach that uses a prioritization based on the Boolean classification (Borner et Paech, 2009b).

Results of the study show that (1) the order plays a key role in early defect discovery and that (2) a trade-off often exists between SBC violations and early defect detection capability. Also, we find that, for a reasonably-good defect prediction, MITER performs better than a random approach in increasing the early defect detection rate while generating substantially less SBC violations. Furthermore, while increasing slightly the number of SBC violations compared to that of orders generated by the traditional CITO approach—only with the objective of minimizing SBC violations—, orders for moderate values of the priority weight outperform orders without prioritization in terms of early defect detection. The empirical study also shows that our approach leads to a better trade-off than the test focus approach.

In the following, we first recall the CITO problem and describe the test focus approach

proposed by Borner et Paech (2009b). Then, we present our formalization of the problem as well as the memetic algorithm that solves it. Finally, we present the design and the results of the empirical study that we performed to evaluate MITER.

5.2 CITO Problem and Existing Approaches

The CITO problem deals with determining an optimal class integration test order, *i.e.*, to devise the order in which classes should be integrated and tested with minimum cost. The cost is measured as the number and–or complexity of stubs to be developed (*i.e.*, SBC violations). As shown by Kung *et al.* (1995), when a class diagram does not contain cycles, an optimal test order is computed with a sort algorithm. Yet, in general, any class diagram of realistic size contains cycles (Briand *et al.*, 2001). Thus, the problem becomes finding an optimal way to eliminate cycles, which is the well-known Feedback Arc Set (FAS) problem. It consists of transforming a Directed Graph (DG), containing cycles, into a Directed Acyclic Graph (DAG) by removing edges to break the cycles while minimizing some cost measures (Eades *et al.*, 1993). The decision version of the FAS problem is one of the 21 NP-complete (Karp, 1972). Therefore, the FAS problem is both NP-hard (Festa *et al.*, 2009): There is no polynomial algorithm to resolve them.

5.2.1 Traditional CITO Approaches

In Section 3.2, we showed that several approaches have been proposed to tackle the CITO problem, some based on a FAS-like formalization (Briand *et al.*, 2002b), some assuming the dependencies among classes to form a DAG (Kung *et al.*, 1995), other proposing a solution more specific to the testing domain (Tai et Daniels, 1997; Hanh *et al.*, 2001; Traon *et al.*, 2000; Abdurazik et Offutt, 2009).

In all these approaches, hereby referred to as traditional CITO approaches, the goal is to minimize the number and–or complexity of SBC violations. While MITER is essentially inspired by previous approaches, it aims at promoting early defect detection in addition to the minimization of SBC violations.

5.2.2 Test Focus Approach

Recently Borner et Paech (2009a,b) proposed an approach to generate class integration test orders with the goal of testing defect-prone classes first and reducing SBC violations. The approach is based on a Boolean classification of classes to test into two sets: (i) the test focus set, *i.e.*, risky classes (defect-prone classes) that must be tested first and (ii) the other

classes (non-test focus set). Then, existing algorithms to derive class integration test order are applied independently on each set with the goal of minimizing SBC violations.

The approach proposed by Borner et Paech (2009a) identifies risky dependencies between classes—hereby referred to as "test focus dependencies"—based on properties of dependencies that correlate with defects occurred in the past. It then includes all classes involved in those dependencies in the test focus set.

To the best of our knowledge, this approach is the only one that takes into account testing effectiveness in terms of defect detection when dealing with the CITO problem. However, the use of the boolean classification does not allow a compromise between the two objectives. Moreover, because classes in the test focus do not have the same degree of defect-proneness, we believe that using a fine-grained prioritization as MITER is more realistic and can lead to better balanced orders.

5.3 Problem Formalization and Algorithm

5.3.1 Problem Formalization

Kung *et al.* (1995) argue that association is the weakest relationship compared to aggregation or inheritance, which involve besides control coupling, code dependency and data coupling. They demonstrate based on the semantics of each relationship that it exists at least one association in each cycle in OO programs. Therefore, their approach allows SBC violations only when the relationship between the client and the server is an association. Such violations yield to build less complex stubs. Many other researchers adopt the same point of view (Tai et Daniels, 1997; Briand *et al.*, 2001) while others choose to not make any distinction between the types of relationships (Traon *et al.*, 2000; Hanh *et al.*, 2001). Malloy *et al.* (2003) propose a more flexible approach (also adopted by MITER) in which a coefficient is assigned to each type of dependency to control whether testers prefer to avoid violations to a particular type of dependency or another.

In the earlier works on the SBC violations, researchers mostly focused on the number of SBC violations. However, this focus has serious limitations and may not be representative of the cost to write stubs. In contrast, Briand *et al.* (2002b) proposed a complexity metric that measures the complexity of the dependency between two classes based on the strength of the coupling between them. MITER can use number or overall complexity of SBC violations, through the parametrization of its cost function.

Let $\mathcal{C}$ be the set of classes to be tested. $\mathcal{C}$ can be all or part of a program or in the context of software evolution, the change impact set.

A solution $S_i =< c_{i,1}, \ldots, c_{i,N} >$ to the problem is an ordered sequence of the N classes of $\mathcal{C}$. Let $Pos(c, S_i)$ be the rank of c in S_i.

Let each class $c \in \mathcal{C} = \{c_1, \ldots, c_N\}$ be associated with a priority $Prio(c)$, *i.e.*, a number between zero and one quantifying the degree of defect-proneness of a class. This priority indicates the desired position in which c should be tested: a class with test priority equals to one means that this class is highly defect-prone and should be tested at the beginning of the testing phase while a priority of zero indicates a class with a very low risk to be faulty and thus testing it can be delayed.

To maximize early defect detection, MITER should minimize the following cost function:

$$F_p(S_i) = \sum_{c \in S_i} Prio(c) \times Pos(c, S_i) \tag{5.1}$$

A class c in $\mathcal{C}$ may act as a server for other classes in $\mathcal{C}$ and–or be a client of other classes in $\mathcal{C}$. Coupling between clients and servers may be due to use relationships (Us), associations (As), aggregations (Ag), compositions (Cp), and–or Inheritances (I).

Let assume that the pair (X, Y) represents the relation between classes X (server class) and Y (client class). $SBC_{S_i} = \{(X, Y) | (X, Y) \in \{Us, As, Ag, Cp, I\} \ \& \ Pos(X, S_i) > Pos(Y, S_i\}$ is then the set of pairs (X, Y) of classes in S_i where a SBC violation is observed.

We define the function $Type(X, Y)$ to return the type of the relationship between X and Y and the function $\gamma(Type(X, Y))$ to impose a penalty, if needed, to favor violating a particular type of relationship rather than another. For example, if $\gamma(As)$ is set to 1 and $\gamma(I)$ is set to 100, then testers favor violating associations rather than inheritances.

Let define $Cpx(X, Y)$ as the complexity measure for the relationship between X and Y. When $Cpx(X, Y)$ is set to 1 for all dependencies, then the goal is to minimize the number of SBC violations.

To minimize SBC violations, MITER should minimize the following cost function:

$$F_s(S_i) = \sum_{(X,Y) \in SBC_{S_i}} \gamma(Type(X, Y)) \times Cpx(X, Y) \tag{5.2}$$

Let further assume that α and β are the weights of the testing class prioritization and the respect of SBC principle, respectively. They model the testers' preference for one objective over the other.

Let assume that $Norm$ is a function that normalizes, in the interval $[0, 1]$, the term represent-

ing the cost of each objective to avoid a dominance of one of the terms due to the possible difference between the ranges of values. The function $Norm$ is defined as $Norm(C_i) = 1 - (1/1 + a \times C_i)$, where C_i is the value to normalize and a is a coefficient that helps to avoid the squeezing of values in the interval $[0, 1]$.

Given the above definitions and considerations, in order to find balanced orders between the two objectives, MITER aims at minimizing the following general cost function:

$$F(S_i) = \alpha Norm(F_p(S_i)) + \beta Norm(F_s(S_i)) \tag{5.3}$$

$F(S_i)$ is a generalisation of the cost functions defined in previous works. Indeed, the four cost functions proposed by Briand *et al.* (2002b) and used in other search-based approaches (Borner et Paech, 2009b; da Veiga Cabral *et al.*, 2010) can be represented using Equation 5.3 as follows:

- Number of broken dependencies (D): $\alpha = 0$, $\gamma(I) = \gamma(Ag) = +\infty$ [1], $\gamma(As) = 1$, $Cpx(X, Y) = 1$ for all dependencies.
- Attribute Coupling (A): $\alpha = 0$, $\gamma(I) = \gamma(Ag) = +\infty$, $\gamma(As) = 1$, $Cpx(X, Y)$ returns only the attribute coupling between X and Y.
- Method Coupling (A): $\alpha = 0$, $\gamma(I) = \gamma(Ag) = +\infty$, $\gamma(As) = 1$, $Cpx(X, Y)$ returns only the method coupling between X and Y.
- Attribute and Method Coupling (Ocplx): $\alpha = 0$, $\gamma(I) = \gamma(Ag) = +\infty$, $\gamma(As) = 1$, $Cpx(X, Y)$ returns weighted geometric average of attribute and method coupling.

In each adaptation of Equation 5.3 shown above, the function $Norm(C_i)$ is the identity function. To cope with other complexities metrics as the one defined in (Abdurazik et Offutt, 2009), it suffices to replace the function $Cpx(X, Y)$ by the right one.

In the following, β is equal to $1 - \alpha$. We then use α to express the balance between the two objectives.

5.3.2 MITER Memetic Algorithm

We propose to solve the problem formalized above using a memetic algorithm because:

1. The CITO problem is an instantiation of the SBC problem and can be modeled as a feedback set problem (FAS). Finding a solution for the FAS problem is NP-hard, therefore metaheuristics approaches may solve it.

1. We set the penalty coefficient higher than the worst possible number of violations.

2. Along the same lines, a comparison of graph-based approaches to metaheuristics-based approaches Briand *et al.* (2002b) showed that genetic algorithm outperforms graph-based approaches in solving the CITO problem.
3. Finally, a preliminary study, presented in Appendix B, compares a memetic algorithm to a genetic algorithm in solving the CITO problem. The results of this experiment show the superiority of the memetic algorithm over the genetic algorithm. Thus, it confirms the effectiveness of memetic algorithms as reported in the literature (Hoos et Sttzle, 2004; Fraser *et al.*, 2015).

In the following, we give an overview of our memetic algorithm and detail its operators.

Overview: A potential solution of the problem formalized above corresponds to an ordering of classes of $\mathcal{C}$ that minimizes the cost function defined 5.3.

The proposed memetic algorithm has two parameters because there is no mutation and at each generation, the crossover is applied to two parents only. These two parameters are the size of the population (*popSize*) and the number of generations (*nbGen*). This minimum number of parameters makes easy parameter tuning—considered as one of the hardest steps in using evolutionary algorithms—without making the algorithm less powerful.

The algorithm starts with a random generation of the initial population. The local search is applied to each individual in the initial population. At each generation, two parents are selected according to a uniform probability. Then, the crossover is applied to the two parents to produce one offspring. The local search (LS) operator is then applied to the offspring. Finally, the offspring replaces the worst individual in the population. The algorithm stops after *nbGen* of generations.

Crossover Operator: The crossover operator used in the memetic algorithm is the position-based crossover proposed by Syswerda (1990) and used by Briand *et al.* (2002b). This operator first randomly selects $N/2$ classes. Each selected class is then copied in the offspring at the same position that it occupied in the first parent. Then, all other classes are used to fill the remaining empty positions in the offspring according to their order in the second parent.

Local Search Operator: The local search operator used in the memetic algorithm is an iterative improvement heuristic also called descent algorithm. It performs repeatedly a move that strictly decreases the cost of the solution and stops when no more improving move is available, *i.e.*, when a local optimum is reached. In our local search heuristic, a move consists in moving the class c_i from its current position i to a new position j ($i \neq j$): such a move is represented by the couple $< i, j >$. Given an ordering $c = (c_1, c_2, \ldots, c_n)$, the solution

$c' = c(+) < i, j >$ denotes the new ordering produced by applying the move $< i, j >$ to c and corresponds to:

– $(c_1, \ldots, c_{i-1}, c_{i+1}, \ldots, c_j, c_i, c_{j+1}, \ldots, c_n)$, if $i < j$;

– $(c_1, \ldots, c_{j-1}, c_i, c_j, c_{j+1}, \ldots, c_{i-1}, c_{i+1}, \ldots, c_n)$, otherwise.

5.4 Study Design

The *goal* of this study is to evaluate MITER with the *purpose* of assessing its ability to support testers in finding a testing order that promotes early defect detection and limits the number of SBC violations. The *perspective* is that of researchers interested to assess the performance of MITER and understand whether it could constitute a valid support for testers when performing class integration testing. Results of our study will also be useful to practitioners interested in determining a cost-effective class integration test order.

The *context* of the study consists of six releases of three open-source programs: Ant, ArgoUML, and Xerces. We chose these programs because they have been the subjects of a previous work in defect location (Bavota *et al.*, 2012). Moreover, they are open-source and they belong to different application domains.

Apache Ant[2] is a build tool for Java, *ArgoUML*[3] an open-source tool for UML diagrams, and *Apache Xerces for Java*[4] a Java XML parser. The present experiment is performed in the context of software evolution. The set $\mathcal{C}$ of classes to be tested are then the classes in the change impact set CIS), *i.e.*, new classes, the classes that have been modified, and also the ones that are impacted by the modified classes.

Table B.1 summarizes information about the program releases considered in the study and, specifically, the total number of classes, the number of new and modified classes[5] (NMC), the size of the change impact set (CIS), the number of defects introduced since the previous release and not fixed, the number of call sites to test, and the number of relations per type. As in previous approaches (Kung *et al.*, 1995; Briand *et al.*, 2002b), we consider three relationships: associations (As), aggregations (Ag), and inheritance (I). Associations include use relationships and aggregations include compositions. For each analyzed program, changes have been identified considering the previous release.

2. http://ant.apache.org/
3. http://argouml.tigris.org/
4. http://xerces.apache.org/xerces2-j/
5. This is a conservative estimate of changes not being detected by Java.

Table 5.1 Study objects: detailed information of the analyzed releases.

Release	Previous Release	Classes	NMC	CIS	Faulty Classes	Nb Defects	Call Sites	As	Ag	I
Ant 1.6.2	1.6.1	623	92	288	19	30	7, 373	2, 136	36	393
Ant 1.7.1	1.7.0	731	209	519	9	14	10 162	2, 719	43	481
ArgoUML 0.14.0	0.12.0	1, 187	802	1, 005	227	473	13, 222	3, 745	21	830
ArgoUML 0.22.0	0.20.0	1, 353	687	939	168	359	24, 973	4 858	25	885
Xerces 2.0.1	2.0.0	396	57	92	18	51	3, 774	1, 058	3	220
Xerces 2.6.2	2.6.1	466	57	89	13	28	4, 263	1, 430	5	268

5.4.1 Research Questions

To answer our main research question, the present study addresses the following two research questions:

- **RQ1**: *What is the effect of class test priority on early defect detection when varying its importance in determining the test order?* We evaluate the effect of class test prioritization on early defect detection with respect to the number of call sites, when α varies from zero (*i.e.*, class priorities are disregarded) to one (*i.e.*, SBC violations disregarded). We expect a higher rate of early defect detection when α increases.
- **RQ2**: *What is the effect of class test priority on the number of server-before-client violations when varying its importance in determining the test order?* We evaluate the effect of class test prioritization on the number of SBC violations when α varies from zero to one. We expect a higher number of SBC violations when α increases. Indeed, if a class A is not ranked in topmost positions when $\alpha = 0$ (*i.e.*, priorities are disregarded), this likely means that A depends upon some server classes that must be tested before it. Therefore, prioritizing A (*i.e.*, moving A in the topmost order positions), may force to test A before its server classes, and thus it increases SBC violations.

In each research question, we will also compare MITER ability against three baselines as described in Section 5.4.2.

For this experiment, we set the parameters of the cost function of MITER (5.3) as follows.

- We favor the violation of associations over aggregations and favor the violation of aggregations over inheritance: $\gamma(As) = 1$; $\gamma(Ag) = 10$;$\gamma(I) = 100$.
- The goal is the minimization of the number of SBC violations: $Cpx(X, Y) = 1$ for all dependencies.
- We set a, the coefficient of the normalization function $Norm$, to 0.001.

We vary α as follows:

– $\alpha = 0$, class prioritization is ignored and the search is only guided by the minimization of the number of SBC violations;
– $\alpha \in \{0.1, 0.25, 0.5, 0.75\}$, the two components are used to guide the search;
– $\alpha = 1$, we consider class prioritization only to guide the search.

In **RQ1**, we assume that if a class is tested, then the defects will be eventually discovered and fixed. This assumption may be invalid as defects may slip through testing. Thus, **RQ1** actually quantifies the upper bound to defect detection; the actual defect detection depends on many factors among which the quality of test data and the skills of the testers. All these relevant factors are out of the scope of this chapter.

5.4.2 Baselines for Comparison

We will compare MITER against three baselines.

1. **Random ordering (*RndOrder*):** We compare MITER against a random class testing ordering. Given $\mathcal{C}$ classes, we pick a random order and compute its number of SBC violations and its early defect detection rate. If MITER does not perform better than a random order, it would be pointless to adopt it.
2. **Optimal ordering (*OptOrder*):** We compare MITER against an "upper-bound" ordering in which classes are ordered in the best possible order in terms of early defect detection and based on a perfect knowledge of the location and the number of defects in each class.
3. **Test focus approach:** We compare MITER against the test focus approach described in Section 5.2.2.

5.4.3 Class Testing Priority

When prioritizing classes to be tested to maximize early defect detection, testers should know to which extent each class is defect-prone. In practice, this information is not available but it is possible to use defect prediction approaches to identify classes that are likely to exhibit a higher number of defects than others (Gyimóthy *et al.*, 2005; Zimmermann et Nagappan, 2008; D'Ambros *et al.*, 2012).

MITER requires defect prediction data. As shown in a recent survey by D'Ambros *et al.* (2012), several kinds of predictors exist based on product and–or process metrics. In this chapter, we rely on a multivariate predictor using linear regression and combining some

product metrics (Chidamber et Kemerer (1994) metrics and LOC) with one process metric (the number of defects introduced in the class before the date of the previous release).

The choice of the product metrics is inspired from a work by Gyimóthy *et al.* (2005) whereas the choice of the process metric is based on a work by Ostrand *et al.* (2005) who performed defect prediction based on previous defects.

Table 5.3 describes the metrics used in our defect prediction model while Table 5.4 reports the precision, recall, and F-measure of the defect predictor. We do not attempt to use the best possible defect prediction approach. Rather, we want to show how MITER can rely on a defect predictor to establish a suitable testing order. Therefore, the more close to the truth is $Prio(c)$, the more effective MITER order will be. Results reported in this chapter can then be considered realistic as we used a well-known and easy to build defect predictor and any improvement in the predictor will translate into a higher early defect detection.

The results of defect prediction are used in our formalization as class testing priorities. We compute the priority value $Prio(c)$ of a class c with the following formula:

$$Prio(c) = \frac{probability(c)}{callsite(c)} \tag{5.4}$$

where $probability(c)$ is the probability of c to be defective as returned by the defect predictor and $callsite(c)$ is the number of c call sites. Given two classes with the same defect probability, this formula gives a higher priority to the smaller one in terms of call sites and, thus, increases the cost-effectiveness: the order will rank first small (low number of call sites) highly defect-prone classes.

Regarding the optimal ordering, we assign to each c in $\mathcal{C}$ its OptOrder priority as follows:

$$Prio(c) = \frac{nbDefects(c)}{maxDefects \times callsite(c)} \tag{5.5}$$

where $nbDefects(c)$ is the number of defects in c and $maxDefects$ is the maximum number of defects in any given class under test. Thus, the *OptOrder* ranks in first positions classes exhibiting the highest ratio of defects per call sites. It is important to underline that this upper bound is not realistic as defects number and location are supposed in this case to be known. Moreover, ordering classes just based on the number of defects per call site disregards SBC violations and potentially leads to orders with extremely high numbers of SBC violations and thus to an unacceptable number of stubs.

Following existing defect prediction approaches (Arisholm *et al.*, 2010; Gyimóthy *et al.*, 2005),

we include in the test focus set each class whose probability to be faulty is equal to or higher than 0.5. We choose to include all classes classified as defect-prone in the test focus and not only classes involved in test focus dependencies because MITER assumes that unit and integration testing activities are performed in parallel. Table 5.2 summarizes for each analyzed release the size of the test focus set and the non-test focus set.

5.4.4 Data Collection

To compute $\mathcal{C}$ in this experiment, we identify relations between classes. To this aim, we use two available tools: the *Ptidej* tools suite [6] and *ChangeDistiller* (Fluri *et al.*, 2007).

The *Ptidej* tools suite extracts facts from Java bytecode or source code and creates PADL models, which include relations between classes and also methods invocations. We use this information to identify server and client classes.

ChangeDistiller detects fine-grained source code changes at the level of statements. We choose among the 40 types of changes provided by Change Distiller, changes whose impact is not detectable at compile time. We select classes with such changes and compute the transitive closure using information about relations between classes and method invocations provided by the PADL model to obtain the set of possibly impacted classes.

To associate defects to a particular release, we rely on the *SZZ* algorithm (Sliwerski *et al.*, 2005; Kim *et al.*, 2006). Based on the *blame/annotate* feature of versioning systems and on a set of heuristics to reduce the noise, *SZZ* identifies when (and by whom) lines changed in a bug fix have been changed the last time before the fix. Although a bug fix can be performed by modifying other lines in the code than those where the bug was introduced, *SZZ* provides a reasonable estimate of the changes that induced a fix and, therefore, when a bug was likely introduced. Therefore, we assume that a defect is associated to release R_i if the earliest fix-inducing change identified by *SZZ* occurs between R_{i-1} and R_i and the bug has been opened after R_i. In this chapter, we use a data set of fix inducing changes already identified for a previous paper (Bavota *et al.*, 2012).

We rely on the tool *POM* [7] of the *Ptidej* tools suite to compute the metrics used to build the defect predictor. POM is a framework that allows computing a wide range of software metrics on PADL models.

6. http://www.ptidej.net/
7. http://wiki.ptidej.net/doku.php?id=pom

Table 5.2 Size of Test Focus (TF) and Non-Test Focus (NTF) Sets.

Release	TF	NTF
Ant 1.6.2	20	278
Ant 1.7.1	8	511
ArgoUML 0.14.0	289	716
ArgoUML 0.22.0	195	744
Xerces 2.0.1	15	77
Xerces 2.6.2	15	74

Table 5.3 Description of the metrics used in the prediction model.

Metrics	Description
CBO	Coupling Between Objects classes
DIT	Depth of Inheritance Tree of a class
LCOM2	Lack of Cohesion in Methods of a class
NOC	Number of Children
RFC	Response for a class
WMC	Weighted Methods Per Class
LOC	Numbers of lines of code
Past Bugs	Number of bugs involving a class in the past

5.4.5 Study Settings

MA settings have been determined through a preliminary experiment. Depending on the size (number of classes) in the set to analyze, we distinguish three groups: small sets of classes are sets that contain at most 100 classes, medium sets are sets that contain more than 100 classes and at most 500 classes. Large sets contain more than 500 classes. Table 5.5 summarizes the different sets of parameters used in the present study for each group.

To account for the randomness of the MA, we execute MITER 20 times for each value of α. We then record the average of the percentages of detected defects (if the classes were tested) and SBC violations as function of the percentages of call sites *i.e.*, for the $i-th$ class in the order, count of call sites for the first i classes.

For the random baseline, we assign the same amount of execution time as the MA to generate $\mathcal{C}$ permutations; then out of the thousands of generated orders ($\mathcal{C}$ permutations) we select the one with fewer SBC violations. We repeat this experiment 20 times.

Table 5.4 Precision, recall and F-measure of logistic Predictors (for classes predicted as defect-prone).

Release	Precision	Recall	F-measure
Ant 1.6.2	30.00	33.33	31.58
Ant 1.7.1	25.00	25.00	25.00
ArgoUML 0.14.0	44.29	56.64	49.71
ArgoUML 0.22.0	47.69	55.69	51.38
Xerces 2.0.1	66.67	58.82	62.50
Xerces 2.6.2	60.00	75.00	66.67

Table 5.5 MA Parameters Settings.

program Type	*popSize*	*nbGens*
Small	30	100
Medium	100	5,000
Large	200	20,000

5.4.6 Analysis Method

To answer **RQ1**, we plot the percentages of detected defects of each order against the percentages of call sites needed to be covered (cost). Each curve plots average values across 20 MA executions (smoothed using Bezier interpolation).

To measure the early defect detection of each order, we adapt the APFD (Average Percentage of Faults Detected) metric defined by Rothermel *et al.* (1999). In the context of test cases prioritization, APFD is defined as the area under each curve and indicates how rapidly defects are detected by the corresponding prioritized test suite. In other words, this area represents the early defect detection rate of each test suite. In our case, we simply call this metric EDDR (Early Defect Detection Rate). Thus, EDDR represents the early defect detection rate of each test order with respect to the cost of testing measured here in terms of call sites. Instead of using the x-axis as bound of the area under curve, we use the curve representing the random order. As explained by Arisholm *et al.* (2010), this allows assessing the early defect detection of a given order to that of the random order. We compute EDDR as follows:

$$EDDR(order)_p = area(order)_p - area(RndOrder)_p \tag{5.6}$$

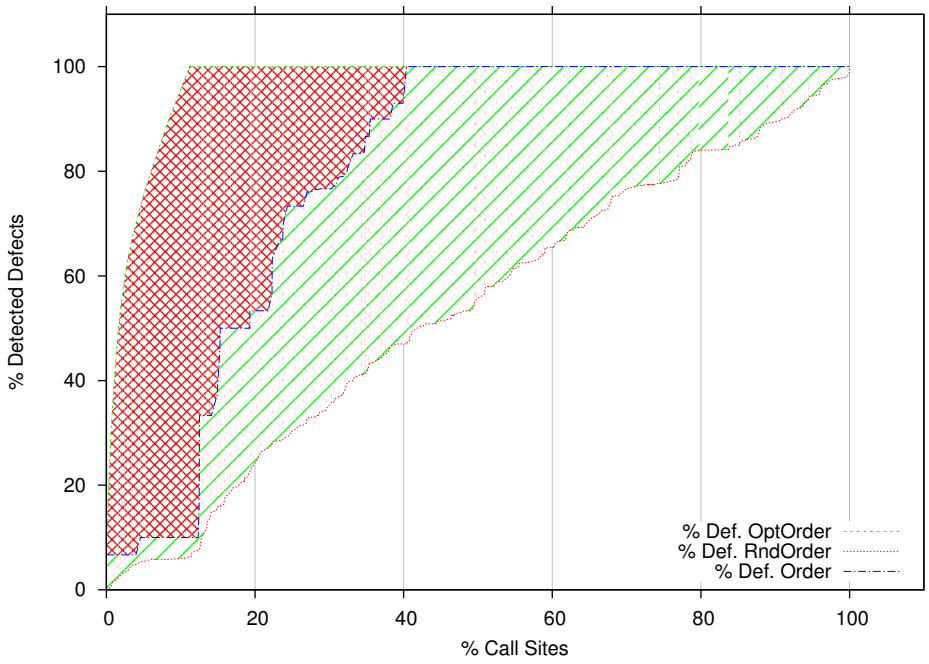


Figure 5.1 Areas Representing the Early Defect Detection Rate of a Given Order (ΛOrder) and the OptOrder compared to RndOrder.

where $area(x)_p$ is the area under the curve x for a given p percentage of call sites (threshold for the cost). A higher and positive EDDR indicates a better order in terms of early defect detection. A negative EDDR actually points to a performance worse than the random order.

To allow a comparison across programs, we further use the normalized EDDR (NEDDR) following the normalization proposed in (Arisholm *et al.*, 2010) and adapted for our purpose as follows:

$$NEDDR(order)_p = \frac{area(order)_p - area(RndOrder)_p}{area(OptOrder)_p - area(RndOrder)_p} \tag{5.7}$$

where $area(x)_p$ is the area under the curve x for a given p percentage of call sites (threshold for the cost), *OptOrder* and *RndOrder* indicate the optimal and random orders, respectively. Because exhaustive testing is usually impossible, we evaluate the *NEDDR(order)* at different call site coverage thresholds (testing budget), namely 20, 40, 60, 80, and 100.

We report the average *NEDDR* achieved by MITER for different weighting of the class priorities, *i.e.*, for different values of α as defined in 5.4.1, and at different call site coverage thresholds.

In the context of **RQ1**, we compare nine types of orders: the six produced by MITER and

corresponding to the different values of α, the test focus order, the random order (RndOrder), and the optimal order (OptOrder).

RQ2 investigates the effect of priorities on the number of *SBC* violations. We collect, for each computed solution, the number and type of SBC violations. Similarly to **RQ1**, we also report the average numbers of SBC violations at different thresholds of call site coverage and different values of α, and compare results with the baselines.

To study the statistical significance of the effect on the *NEDDR* and *SBC* violations of different α values, of different threshold values, and of the program releases on which the study was conducted, we use permutation tests on data of each run. We use an implementation available in the *lmPerm* R package. We set the number of iterations of the permutation test procedure to 500,000. Since the permutation test samples permutations of combinations of factor levels, multiple runs of the test may produce different results, we choose a high number of iterations such that results do not vary over multiple executions of the procedure.

Besides the above analysis, we study the statistical significance of the difference between MITER orders (for different values of α and for different thresholds) and test focus order using the Mann-Whitney two-tailed test in both research questions at different threshold of coverage. We perform the test using the data of each run.

Our null hypotheses for **RQ1** and **RQ2** are respectively:

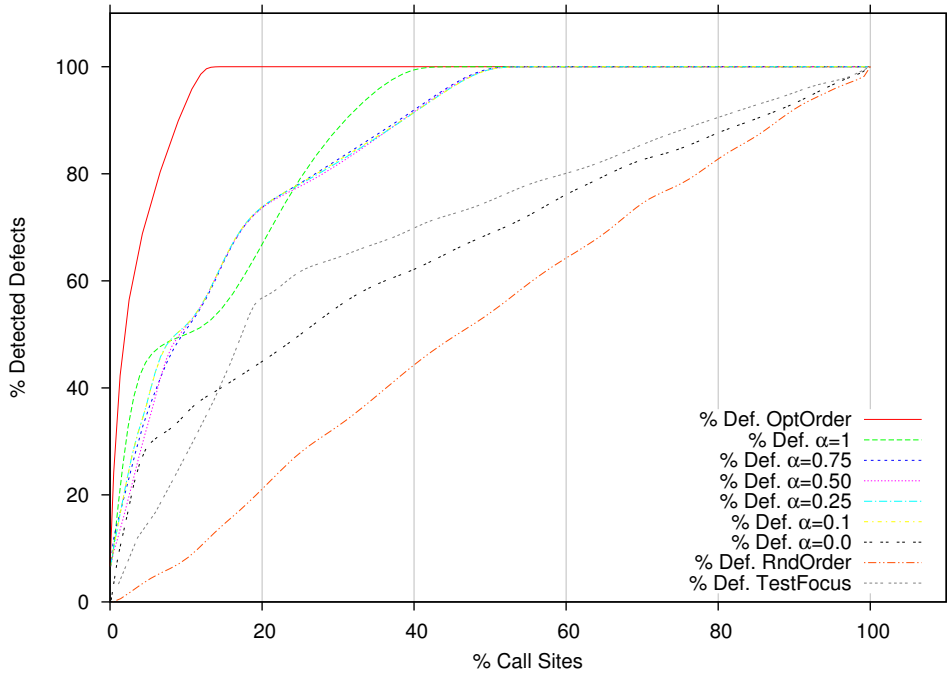
– H_{01}: There is no significant difference between the NEDDR of MITER order (for a given α) and test focus order.
– H_{02}: There is no significant difference between the number of SBC violations of MITER order (for a given α) and test focus order.

We also estimate, using the non-parametric Cliff's d (Grissom et Kim, 2005) effect size measure, the magnitude of the differences of means of NEDDR and number of SBC violations between MITER order (for a given α) and the test focus order.

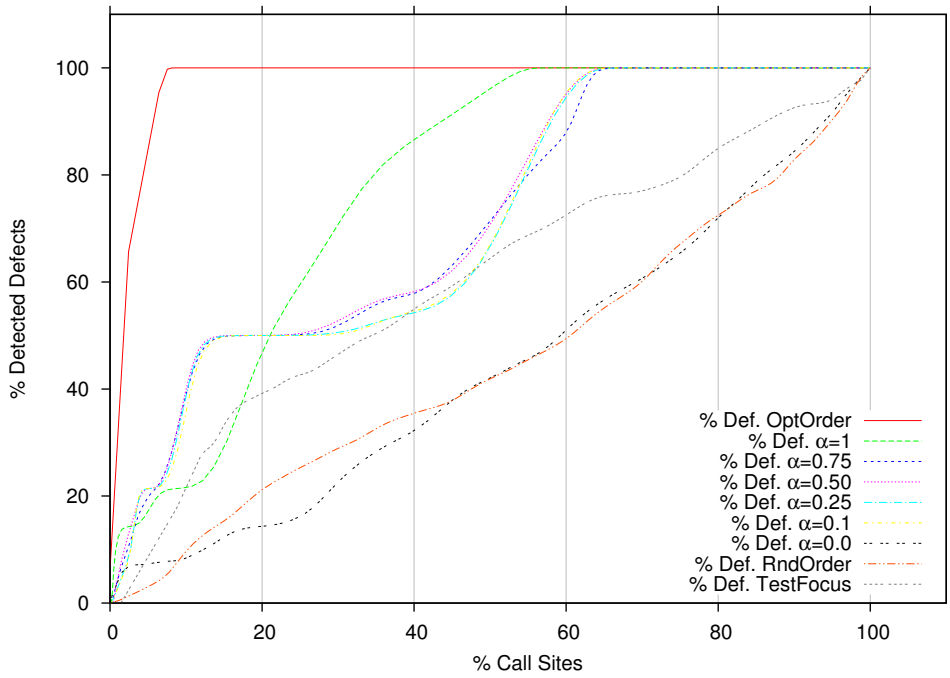
Because the above described analysis requires multiple comparisons, we adjust p-values using Holm's correction procedure (Holm, 1979).

5.5 Results and Discussion

This section reports the study results.

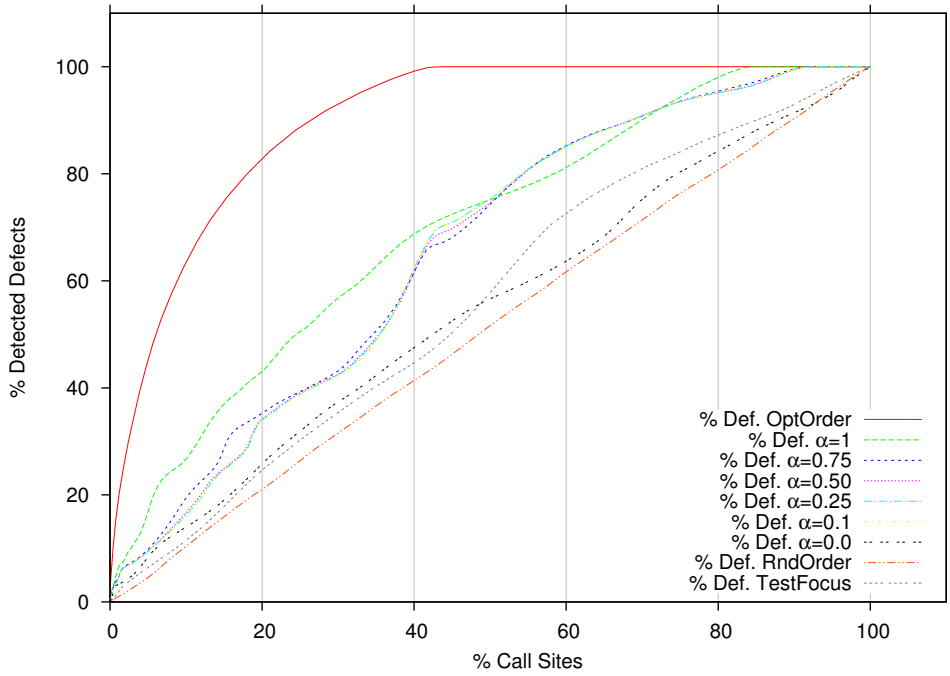


(a) Ant 1.6.2

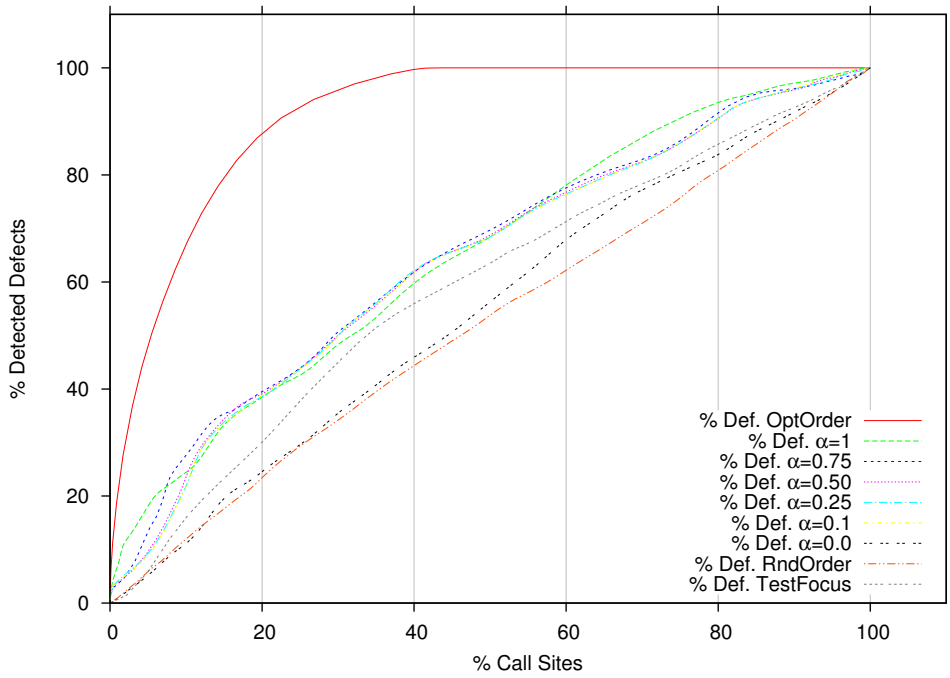


(b) Ant 1.7.1

Figure 5.2 Defect Detection Rate Curves of MITER Orders, RndOrder, OptOrder, and Test Focus Order.

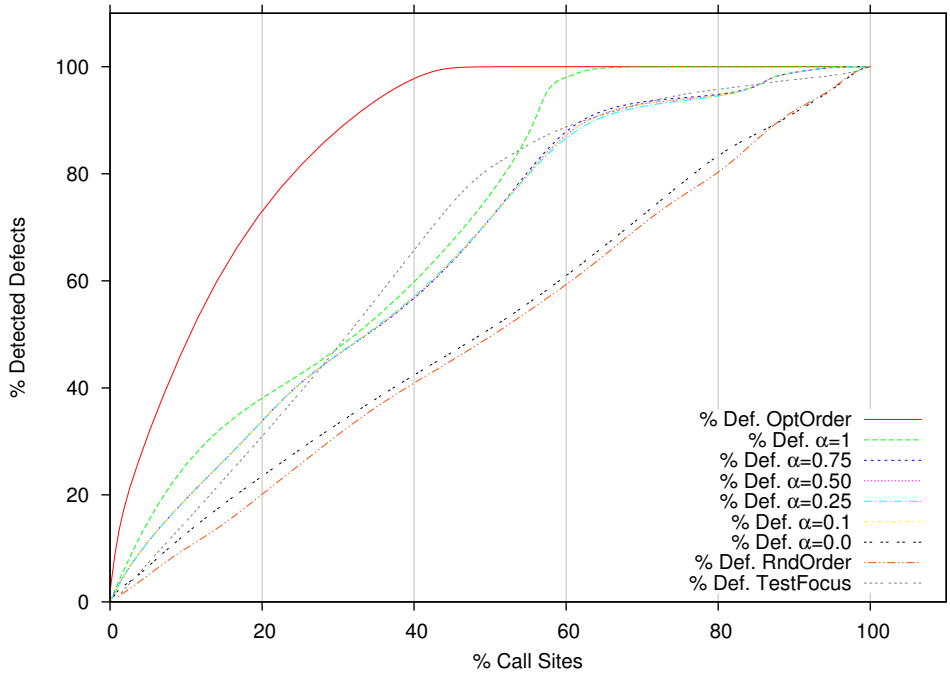


(a) ArgoUML 0.14.0

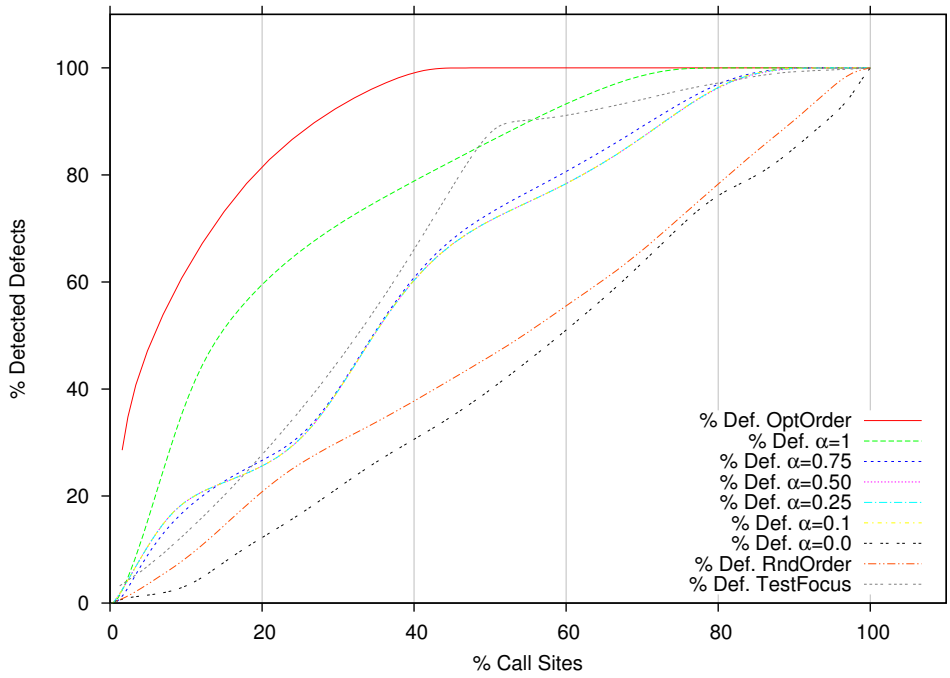


(b) ArgoUML 0.22.0

Figure 5.3 Defect Detection Rate Curves of MITER Orders, RndOrder, OptOrder, and Test Focus Order.



(a) Xerces 2.0.1



(b) Xerces 2.6.2

Figure 5.4 Defect Detection Rate Curves of MITER Orders, RndOrder, OptOrder, and Test Focus Order.

Table 5.6 Ant - Average NDDRE for Different α and Coverage Thresholds.

Call Sites (%)		20	40	60	80	100
α	0	0.31	0.32	0.32	0.32	0.31
	0.1	0.53	0.62	0.71	0.75	0.76
	0.25	0.53	0.63	0.71	0.75	0.76
	0.5	0.52	0.61	0.69	0.74	0.75
	0.75	0.52	0.63	0.71	0.75	0.75
	1	0.49	0.65	0.73	0.78	0.78
TestFocus		0.25	0.35	0.37	0.38	0.39
RndOders (area)		174.62	841.28	1929.14	3389.82	5233.05
OptOrder (area)		1676.91	3692.37	5699.69	7669.04	9700.78

(a) Ant 1.6.2

Call Sites (%)		20	40	60	80	100
α	0	0.00	-0.04	-0.03	-0.03	-0.02
	0.1	0.29	0.30	0.34	0.45	0.48
	0.25	0.30	0.31	0.34	0.45	0.48
	0.5	0.31	0.32	0.37	0.48	0.51
	0.75	0.30	0.32	0.37	0.47	0.49
	1	0.18	0.37	0.52	0.60	0.62
TestFocus		0.13	0.19	0.24	0.27	0.28
RndOrder (area)		188.13	759.82	1603.50	2812.26	4500.89
OptOrder (area)		1803.99	3823.28	5829.77	7799.86	9832.92

(b) Ant 1.7.1

Table 5.7 ArgoUML - Average NDDRE for Different α and Coverage Thresholds.

Call Sites (%)		20	40	60	80	100
α	0	0.08	0.08	0.09	0.09	0.09
	0.1	0.15	0.17	0.27	0.33	0.36
	0.25	0.16	0.18	0.27	0.34	0.36
	0.5	0.16	0.18	0.28	0.34	0.36
	0.75	0.21	0.20	0.29	0.35	0.37
	1	0.37	0.37	0.41	0.42	0.47
TestFocus		0.04	0.05	0.08	0.12	0.12
RndOrder (area)		205.84	834.65	1858.82	3275.83	5101.52
OptOrder (area)		1055.16	3027.33	5016.93	6994.69	9032.21

(a) ArgoUML 0.14.0

Call Sites (%)		20	40	60	80	100
α	0	0.01	0.01	0.03	0.05	0.05
	0.1	0.19	0.25	0.25	0.28	0.29
	0.25	0.20	0.25	0.26	0.28	0.29
	0.5	0.21	0.26	0.26	0.28	0.30
	0.75	0.25	0.28	0.28	0.30	0.32
	1	0.25	0.27	0.25	0.29	0.33
TestFocus		0.06	0.13	0.15	0.16	0.17
RndOrder (area)		233.46	919.21	1986.50	3409.27	5224.92
OptOrder (area)		1241.75	2933.41	5167.88	7135.21	9172.61

(b) ArgoUML 0.22.0

Table 5.8 Xerces - Average NDDRE for Different α and Coverage Thresholds.

Call Sites (%)		20	40	60	80	100
α	0	0.08	0.05	0.02	0.04	0.05
	0.1	0.24	0.26	0.26	0.39	0.40
	0.25	0.24	0.25	0.27	0.39	0.40
	0.5	0.24	0.26	0.26	0.39	0.40
	0.75	0.24	0.26	0.24	0.39	0.40
	1	0.34	0.30	0.37	0.49	0.49
TestFocus		0.11	0.11	0.34	0.42	0.43
RndOrder (area)		170.99	807.62	1803.55	3196.25	5039.70
OptOrder (area)		936.13	2748.32	4693.21	6646.04	8792.30

(a) Xerces 2.0.1

Call Sites (%)		20	40	60	80	100
α	0	-0.04	-0.13	-0.13	-0.14	-0.17
	0.1	0.07	0.08	0.20	0.32	0.34
	0.25	0.07	0.08	0.20	0.32	0.34
	0.5	0.07	0.08	0.20	0.32	0.34
	0.75	0.15	0.09	0.27	0.32	0.37
	1	0.50	0.53	0.57	0.69	0.71
TestFocus		0.13	0.14	0.32	0.40	0.43
RndOrder (area)		143.43	766.79	1648.39	2989.14	4830.28
OptOrder (area)		1009.31	3016.07	5075.65	7008.56	9082.22

(b) Xerces 2.6.2

Table 5.9 Ant 1.6.2 - Comparing NDDRE of MITER and Test Tocus: Mann-Whitney Test Adjusted p-values (p), Cliff's d Effect Size (d) and Magnitude (m) (L: Large, M: Medium, S:Small, N: Negligible).

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	0.33	M	0.07	-0.18	S	0.34	-0.26	S	0.17	-0.27	S	0.16	-0.31	S	0.10
0.1	0.95	L	**0.00**	0.94	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**
0.25	0.95	L	**0.00**	0.94	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**
0.5	0.94	L	**0.00**	0.93	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**
0.75	0.94	L	**0.00**	0.94	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**
1	0.93	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**

Table 5.10 Ant 1.7.1 - Comparing NDDRE of MITER and Test Focus: Mann-Whitney Test Adjusted p-values (p), Cliff's d Effect Size (d) and Magnitude (m) (L: Large, M: Medium, S:Small, N: Negligible).

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	-0.73	L	**0.00**	-0.69	L	**0.00**	-0.67	L	**0.00**	-0.66	L	**0.00**	-0.62	L	**0.00**
0.1	0.86	L	**0.00**	0.54	L	**0.01**	0.40	M	0.07	0.56	L	**0.00**	0.60	L	**0.00**
0.25	0.90	L	**0.00**	0.54	L	**0.01**	0.35	M	0.07	0.56	L	**0.00**	0.58	L	**0.00**
0.5	0.93	L	**0.00**	0.64	L	**0.00**	0.49	L	**0.03**	0.66	L	**0.00**	0.66	L	**0.00**
0.75	0.91	L	**0.00**	0.63	L	**0.00**	0.48	L	**0.03**	0.60	L	**0.00**	0.64	L	**0.00**
1	0.38	M	**0.05**	0.75	L	**0.00**	0.88	L	**0.00**	0.91	L	**0.00**	0.91	L	**0.00**

Table 5.11 ArgoUML 0.14.0 - Comparing NDDRE of MITER and Test Focus: Mann-Whitney Test Adjusted p-values (p), Cliff's d Effect Size (d) and Magnitude (m) (L: Large, M: Medium, S:Small, N: Negligible).

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	0.46	M	**0.02**	0.49	L	**0.01**	0.20	S	0.29	-0.30	S	0.10	-0.35	M	0.06
0.1	0.87	L	**0.00**	1.02	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**
0.25	0.97	L	**0.00**	0.90	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**
0.5	0.89	L	**0.00**	0.92	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**
0.75	0.94	L	**0.00**	0.94	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**
1	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**

Table 5.12 ArgoUML 0.22.0 - Comparing NDDRE of MITER and Test Focus: Mann-Whitney Test Adjusted p-values (p), Cliff's d Effect Size (d) and Magnitude (m) (L: Large, M: Medium, S:Small, N: Negligible).

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	-0.54	L	**0.00**	-0.88	L	**0.00**	-0.99	L	**0.00**	-0.89	L	**0.00**	-0.83	L	**0.00**
0.1	0.90	L	**0.00**	1.00	L	**0.00**	0.88	L	**0.00**	0.87	L	**0.00**	0.98	L	**0.00**
0.25	0.90	L	**0.00**	1.00	L	**0.00**	0.98	L	**0.00**	0.88	L	**0.00**	0.97	L	**0.00**
0.5	0.94	L	**0.00**	0.93	L	**0.00**	0.93	L	**0.00**	0.90	L	**0.00**	1.00	L	**0.00**
0.75	0.95	L	**0.00**	0.94	L	**0.00**	1.02	L	**0.00**	1.02	L	**0.00**	1.02	L	**0.00**
1	0.95	L	**0.00**	0.93	L	**0.00**	1.01	L	**0.00**	0.91	L	**0.00**	0.94	L	**0.00**

Table 5.13 Xerces 2.0.1 - Comparing NDDRE of MITER and Test Focus: Mann-Whitney Test Adjusted p-values (p), Cliff's d Effect Size (d) and Magnitude (m) (L: Large, M: Medium, S:Small, N: Negligible).

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	-0.19	S	0.31	-0.33	M	0.10	-0.90	L	**0.00**	-0.84	L	**0.00**	-0.85	L	**0.00**
0.1	0.57	L	**0.01**	0.43	M	0.10	-0.43	M	0.10	-0.06	N	1.00	-0.18	S	1.00
0.25	0.57	L	**0.01**	0.43	M	0.10	-0.39	M	0.11	-0.09	N	1.00	-0.18	S	1.00
0.5	0.57	L	**0.01**	0.43	M	0.10	-0.39	M	0.11	-0.07	N	1.00	-0.18	S	1.00
0.75	0.57	L	**0.01**	0.43	M	0.10	-0.42	M	0.10	-0.06	N	1.00	-0.18	S	1.00
1	0.91	L	**0.00**	0.59	L	**0.01**	0.24	S	0.19	0.14	N	1.00	0.24	S	1.00

Table 5.14 Xerces 2.6.2 - Comparing NDDRE of MITER and Test Focus: Mann-Whitney Test Adjusted p-values (p), Cliff's d Effect Size (d) and Magnitude (m) (L: Large, M: Medium, S:Small, N: Negligible).

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	-0.36	M	0.25	-0.55	L	**0.01**	-0.89	L	**0.00**	-0.92	L	**0.00**	-0.92	L	**0.00**
0.1	-0.05	N	1.00	-0.09	N	1.00	-0.43	M	0.07	-0.27	S	0.63	-0.34	M	0.27
0.25	-0.05	N	1.00	-0.09	N	1.00	-0.43	M	0.07	-0.27	S	0.63	-0.34	M	0.27
0.5	-0.05	N	1.00	-0.09	N	1.00	-0.43	M	0.07	-0.27	S	0.63	-0.34	M	0.27
0.75	0.23	S	0.88	-0.07	N	1.00	-0.35	M	0.07	-0.21	S	0.63	-0.26	S	0.27
1	0.91	L	**0.00**	0.94	L	**0.00**	0.83	L	**0.00**	0.85	L	**0.00**	0.90	L	**0.00**

Table 5.15 Permutation Test on NDDRE.

	Df	R Sum Sq	R Mean Sq	Pr(Prob)
program	5	71.16	14.23	0.00
th	1	11.50	11.50	0.00
program:coverage	5	1.28	0.26	0.00
approach	5	63.05	12.61	0.00
program:approach	25	13.15	0.53	0.00
coverage:approach	5	3.41	0.68	0.00
program:coverage:approach	25	1.87	0.07	0.00
Residuals	3,528	86.23	0.02	

Table 5.16 Classes with Non-Null Priority.

Release	Classes	Classes ($Prio(c) > 0$)
Ant 1.6.2	288	92
Ant 1.7.1	519	209
ArgoUML 0.14	1,005	802
ArgoUML 0.22	939	687
Xerces 2.0.1	92	57
Xerces 2.6.2	89	57

5.5.1 RQ1: What is the effect of class test priority on early defect detection when varying its importance in determining the test order?

Figures 5.2 to 5.4 report the defect detection curves for MITER (different α values), test focus, random, and optimal for the analyzed programs.

The graphs show the proportion of defects potentially detected (y-axis) as function of call sites coverage (x-axis) if testing classes in the order represented by the curve. The two extremes are represented by *RndOrder*, the random order of classes (45% line, *i.e.*, no prioritization and no minimization of SBC violations) and by *OptOrder*, the optimal upper bound in terms of early defect detection. In all programs, MITER is always better than a random ordering. For moderately high values of α, MITER is equal to or better than the test focus. The behavior is however different: while in Ant, there is a step increase at the beginning of the testing activity followed by a more slow defect discovery rate, in the other analyzed programs the trend is almost reversed.

We can explain the reasons of this difference by the low proportion of faulty classes in Ant. Indeed, only about 7% of classes are faulty in Ant 1.6.2 and about 2% in Ant 1.7.1 while the proportion of faulty classes range between 14% and 23% in the other programs.

Therefore, the percentage of call sites (cost) to test all faulty classes is reduced in Ant: 11% in Ant 1.6.2 and 7% in Ant 1.7.1 while it is about 40% in the other programs. In Ant, all defects are detected before 15% of call sites while it is around 40% in the other programs (see Figs. 5.3–5.4). Moreover, in Ant, at least 20% of faulty classes have 0 or 1 dependency: those classes are ranked in the top classes, leading to a high defect detection rate (at least 20%) at the very beginning of the testing activity.

Tables 5.6–5.8 summarize the normalized early defect detection rate (NEDDR) for each type of order at different coverage thresholds for all analyzed programs. Tables 5.6–5.8 are divided in two parts: the upper parts report NEDDR values for various configurations; the lower parts (OptOrder and RndOrder) report the area used in the calculation of the top part values via Equation 5.7. NEDDR values show that it is important to account for the probability of a class to be faulty. Even for moderately high values of α, MITER outperforms the test focus approach as well as the traditional CITO approach, *i.e.*, $\alpha = 0$, see below.

Table 5.15 reports the permutation test results of the effect of different α values, threshold values, and program releases on NEDDR values. The results of the Mann-Whitney and Cliff's d tests, which compute the differences between MITER and test focus orders, are summarized in Tables 5.9–5.14. Statistically significant p-values (*i.e.*, $p < 0.05$) are reported in bold face. As shown by Table 5.15, there are interactions between the program, the call site coverage (*i.e.*, coverage) and the type of order (*i.e.*, approach) with a very strong significance level. Therefore, we expect that results vary when varying the program, the required coverage, and the type of order.

Traditional CITO Approach (MITER results without prioritization, $\alpha = 0$): Without prioritization, MITER results are close to the 45% line and are, in most releases, slightly above the random order except for three program: Ant 1.7.1 and Xerces 2.6.2, where they are below, and Ant 1.6.2, where the MITER curve stays well above. The NEDDR values support these observations: except in Ant 1.6.2, where the NEDDR is always greater than 0.30 for the different thresholds of coverage, it is less than 0.1 in all the analyzed releases and even negative for Ant 1.7.1 and Xerces 2.6.2. Those results show that traditional CITO approach does not help improve early defect detection.

MITER results with prioritization ($0 < \alpha <= 1$): In all the analyzed releases, MITER orders with $\alpha > 0$ perform better than random orders: the NEDDR values are all positive. As shown in Figures 5.2 to 5.4, when we consider a very small increase of α, in all the analyzed releases at 20% of coverage, there is an increase of the early defect detection rate of at least 10% for MITER orders compared to that of MITER orders without prioritization.

The corresponding NEDDR values increase by at least 0.15 compared to the case of $\alpha = 0$ in most releases except ArgoUML 0.14.0 and Xerces 2.6.2. For example, in Ant 1.7.1, the NEDDR value at 20% of coverage is 29 while it is zero for the order without prioritization. This translates into 50% against 15% in terms of defect detection rates.

Surprisingly, when α varies from 0.1 to 0.75, the NEDDR values for a given coverage threshold are very close to each other: the corresponding curves in the graphs are almost confounded. For example, the NEDDR values are between 52 and 53 at 20% of coverage in Ant 1.6.2. This observation suggests a small contribution of the maximization of early defect detection in the overall cost. However, when checking the specific orders for the different values of α, we observe that the orders are different. They show an effective contribution of the maximization of early defect detection in the overall cost. Indeed, the more α grows, more classes with a high priority are pushed into the top ranks. This trend is not visible in the NEDDR values or on the curves (or very slightly) because of the quality of the predictor.

Among classes with high priority, there are some non-faulty classes that are pushed in the top ranks when α grows and, therefore, the early defect detection does not increase as expected and can even decrease. For example, Table 5.17 describes for each release the distribution of faulty and non-faulty classes based on priority values. We can see in this table that the number of non-faulty classes in ArgoUML 0.14.0 with a priority higher than the third quartile is 156, while the number of actual faulty classes for that quartile is only 45. Moreover, there are also faulty classes with null or very low priority that will not be ranked earlier. As we can see in Table 5.17, there are two faulty classes in ArgoUML with a null priority and one third of the faulty classes in Ant 1.6.2 has a priority lower than the first quartile.

The same explanation holds when $\alpha = 1$, *i.e.*, the search is only guided by class prioritization. the NEDDR is not that much higher than the cases where $\alpha < 1$. Another interesting observation is that, sometimes, orders with higher values of α perform less well than orders with lower values of α. In Ant 1.7.1 between 0% and 20% of coverage where the NEDDR of orders with α between 0.1 and 0.75 are significantly higher than that of the order with $\alpha = 1$ (the curves of orders with α between 0.1 and 0.75 are above that of MITER order with $\alpha = 1$ in Figure 5.2b). We can also explain this situation by the quality of the predictor. Indeed, when α grows, class prioritization gains more and more importance in the overall cost, forcing MITER to rank classes with the highest priority values in the top. Consequently, false positive defect-prone classes are also ranked in the top. When $\alpha = 1$, the search is only guided by class prioritization, thus MITER will rank all classes with non-null priority in the top ranks. These results suggest that MITER results highly depend on the quality of the predictor used to assign priorities to classes. Thus, the better the predictor the more MITER

will improve early defect detection and produce orders that will be close to the optimal when α grows.

In most releases, the NEDDR increases even slightly with the coverage showing that the performances of MITER orders increase with coverage in comparison to random orders. Thus, the NEDDR at 40% is generally higher than that at 20%. For example, in ArgoUML 0.22.0, the NEDDR at 20% of coverage is 0.15 while at 40% of coverage, it is 0.19.

Overall, these results show that taking into account class prioritization (even with a very small weight, for example, $\alpha = 0.1$) can considerably increase the early defect detection rate compared to that of MITER orders without prioritization and random orders. However, the effect of class prioritization on early defect detection is influenced by several factors including the quality of the defect predictor, the number of faulty classes, and the number of defects and their distribution across the program.

Comparison between MITER and Test Focus Orders: Test focus orders perform better than random orders: as Tables 5.6–5.9b show, their NEDDR values are always greater than zero. Tables 5.9–5.12 show that in most analyzed releases, MITER outperforms the test focus approach with a large effect size. No matter the value of α between 0.1 and 0.75 in Ant and ArgoUML (and for different coverage thresholds), MITER achieves much better results than test focus. For example, in Ant 1.6.2, the NEDDR values achieved by the test focus range between 0.25 and 0.39, while for MITER they range between 0.53 and 0.78. However, on Xerces, the test focus performs slightly better than MITER except before 40% in Xerces 2.0.1 where MITER orders are better (cf. Table 5.9a) but the differences are not statistically significant. In all analyzed releases, except in Ant 1.6,2, test focus allows discovering defects earlier, *i.e.*, performs better than traditional CITO (MITER with $\alpha = 0$) orders, as expressed by the negative values of the effect size.

The reason why in general test focus performs less well than MITER with $\alpha > 0$ is related to the sets focus that contain only classes with a faulty probability equal to or greater than 0.5. The number of classes in the test focus sets are close to the number of actual faulty classes. When analyzing these classes in ArgoUML and Ant releases, we observe that most of them are false positives. For example, in Ant 1.6.2, 70% of classes in the test focus set are false positives (see Table 5.18). Thus, since the test focus approach does not make any difference between classes in the test focus in terms of probability of being faulty, it ranks them just based on the SBC violations minimization. As a result, the early defect detection rate is then highly deteriorated by those false positives. If classes with defect probability equal to or greater than 0.5 are mostly true positives as in Xerces releases (for example, in Xerces 2.6.2, two third of the classes are true positives), the test focus performs better. Thus, the

test focus approach seems to be more influenced by the quality of the predictor. Nonetheless, in all studied releases, MITER with $\alpha = 1$ performs better than test focus at all coverage thresholds (except in Xerces 2.0.1 after 40% of coverage). The differences are significant with large magnitude of the effect size.

RQ1 Summary: *MITER is effective to support early defect detection through testing, even if the weight given to class priority (α) is small. MITER with $\alpha > 0$ significantly outperforms test focus.*

5.5.2 RQ2: What is the effect of class test priority on the number of server-before-client violations when varying its importance in determining the test order?

MITER strives to promote early defect detection keeping low the number of SBC violations. Tables 5.19 to 5.21 report, for the different coverage thresholds and for each type of order (MITER, test focus, random, and optimal) the average number of SBC violations. The types of violations are summarized in Table 5.29 while the statistical comparisons between MITER orders and test focus orders are reported in Tables 5.22–5.27.

Traditional CITO approach (MITER results without prioritization, $\alpha = 0$):

Traditional CITO approach has the lowest number of SBC violations when 100% of coverage is achieved; It is then the reference for comparisons with other orders. It may happen that for values of coverage lower than 100%, Traditional CITO orders with $\alpha = 0$ have not the lowest possible number of SBC violations.
The interplay between defect probability and SBC lead to situations where, with $\alpha > 0$ and with a partial coverage, MITER finds orders with a lower number of SBC violations.

As shown in the tables, the number of violations for the two extreme orders, optimal and random, are very high for all analyzed releases and for all coverage thresholds. In Ant 1.6.2, the average number of SBC violations varies from 152 to 400 for random order, and from

Table 5.17 Distribution of Faulty (F) and Non-Faulty (NF) Classes Based on Priorities Values Quartiles (Q).

Release	$Prio(c) = 0$		$0 < Prio(c) <= 1stQ$		$1stQ < Prio(c) <= 2ndQ$		$2ndQ < Prio(c) <= 3rdQ$		$Prio(c) > 3rdQ$	
	F	NF	F	NF	F	NF	F	NF	F	NF
Ant 1.6.2	0	196	6	17	5	18	3	20	5	18
Ant 1.7.1	0	310	4	49	2	50	1	51	2	50
ArgoUML 0.14.0	0	203	82	119	37	163	63	137	45	156
ArgoUML 0.22.0	2	252	71	101	36	136	30	141	31	141
Xerces 2.0.1	0	35	1	14	2	12	6	8	9	5
Xerces 2.6.2	0	32	6	9	5	9	2	12	0	14

Table 5.18 True (T) and False (F) Positive Classes in Test Focus Set.

Release	T	F
Ant 1.6.2	6	14
Ant 1.7.1	2	6
ArgoUML 0.14.0	129	160
ArgoUML 0.22.0	94	101
Xerces 2.0.1	11	4
Xerces 2.6.2	10	5

Table 5.19 Average SBC Violations for Different Coverage in Ant Versions.

Call Sites (%)		20	40	60	80	100
α	0	21.90	25.20	27.15	27.90	28.00
	0.1	23.00	23.00	25.35	27.35	28.00
	0.25	23.00	23.45	25.55	27.15	28.00
	0.5	26.00	26.00	28.50	30.10	31.00
	0.75	35.00	35.65	37.35	39.35	40.00
	1	166.00	225.45	305.45	346.85	361.50
TestFocus		114.35	119.90	122.80	123.65	124.00
RndOrder		152.40	269.15	342.65	385.50	400.75
OptOrder		166.00	324.00	421.00	464.00	471.00

(a) Ant 1.6.2

Call Sites (%)		20	40	60	80	100
α	0	43.80	45.80	47.40	48.55	49.20
	0.1	35.90	44.80	48.45	50.70	51.75
	0.25	37.95	45.95	49.50	51.25	52.80
	0.5	44.55	53.90	57.45	59.40	60.90
	0.75	77.85	88.85	93.60	99.55	100.85
	1	393.30	594.75	744.55	896.90	946.10
TestFocus		130.20	137.20	138.85	139.80	140.35
RndOrder		393.20	685.95	862.55	960.15	992.90
OptOrder		417.00	761.00	917.00	1003.00	1037.00

(b) Ant 1.7.1

Table 5.20 Average SBC Violations for Different Coverage in ArgoUML Versions.

Call Sites (%)		20	40	60	80	100
α	0	82.90	91.75	97.15	97.65	98.70
	0.1	95.40	102.90	104.85	106.55	108.70
	0.25	95.55	103.00	105.00	106.50	109.00
	0.5	97.20	104.95	107.05	108.65	111.05
	0.75	105.55	115.35	117.30	119.30	121.30
	1	872.45	1305.70	1596.00	1681.85	1789.55
TestFocus		277.05	580.65	870.70	874.35	876.65
RndOrder		650.65	1132.85	1457.60	1640.60	1697.85
OptOrder		726.00	1071.00	1435.00	1693.00	1812.00

(a) ArgoUML 0.14.0

Call Sites (%)		20	40	60	80	100
α	0	62.75	85.50	98.70	105.25	106.90
	0.1	89.80	99.50	108.95	113.90	118.00
	0.25	90.60	100.00	109.55	114.25	118.75
	0.5	100.10	107.85	117.75	122.55	127.05
	0.75	122.85	133.60	144.65	148.25	152.45
	1	1039.55	1442.30	1609.10	1676.30	1731.80
TestFocus		393.15	797.15	1036.30	1047.50	1051.15
RndOrder		688.70	1167.05	1474.35	1659.30	1725.50
OptOrder		701.00	915.00	1467.00	1853.00	2061.00

(b) ArgoUML 0.22.0

Table 5.21 Average SBC Violations for Different Coverage in Xerces Versions.

Call Sites (%)		20	40	60	80	100
α	0	8.10	17.30	22.55	25.30	26.00
	0.1	8.00	24.00	26.00	26.00	26.00
	0.25	8.00	24.00	26.00	26.00	26.00
	0.5	8.00	24.00	26.00	26.00	26.00
	0.75	8.00	24.00	26.00	26.00	26.00
	1	40.00	94.00	147.00	161.90	164.55
TestFocus		26.35	54.40	67.65	69.45	71.00
RndOrder		40.55	75.95	99.75	118.35	124.75
OptOrder		48.00	71.00	116.00	130.00	133.00

(a) Xerces 2.0.1

Call Sites (%)		20	40	60	80	100
α	0	11.65	17.10	25.75	31.40	32.00
	0.1	15.00	20.00	20.00	32.00	32.00
	0.25	15.00	20.00	20.00	32.00	32.00
	0.5	15.00	20.00	20.00	32.00	32.00
	0.75	18.00	23.00	27.00	35.25	35.25
	1	35.00	51.00	65.00	73.35	80.30
TestFocus		24.40	40.95	52.30	56.65	58.00
RndOrder		31.85	59.70	79.90	92.05	95.65
OptOrder		38.00	57.00	103.00	115.00	119.00

(b) Xerces 2.6.2

Table 5.22 Ant 1.6.2 - Mann-Whitney Test on SBC Violations for MITER Orders vs Test Focus Order.

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.1	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.25	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.5	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.75	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
1	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**

Table 5.23 Ant 1.7.1 - Mann-Whitney Test on SBC Violations for MITER Orders vs Test Focus Order.

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.1	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.25	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.5	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.75	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
1	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**

Table 5.24 ArgoUML 0.14.0 - Mann-Whitney Test on SBC Violations for MITER Orders vs Test Focus Order.

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.1	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.25	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.5	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.75	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
1	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**

Table 5.25 ArgoUML 0.22.0 - Mann-Whitney Test on SBC Violations for MITER Orders vs Test Focus Order.

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.1	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.25	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.5	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.75	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
1	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**

Table 5.26 Xerces 2.0.1 - Mann-Whitney Test on SBC Violations for MITER Orders vs Test Focus Order.

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.1	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.25	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.5	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.75	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
1	0.80	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**

Table 5.27 Xerces 2.6.2 - Mann-Whitney Test on SBC Violations for MITER Orders vs Test Focus Order.

Call Sites (%) / α	20			40			60			80			100		
	d	m	p	d	m	p	d	m	p	d	m	p	d	m	p
0	-0.90	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.1	-0.80	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.25	-0.80	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.5	-0.80	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
0.75	-0.45	M	**0.01**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**	-1.00	L	**0.00**
1	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**	0.95	L	**0.00**

Table 5.28 Permutation Test on SBC Violations.

	Df	R Sum Sq	R Mean Sq	Pr(Prob)
program	5	76319520.54	15263904.11	0.00
coverage	1	7225105.07	7225105.07	0.00
program:coverage	5	4204922.54	840984.51	0.00
approach	7	327305333.35	46757904.76	0.00
program:approach	35	242766373.93	6936182.11	0.00
coverage:approach	7	22277413.78	3182487.68	0.00
program:coverage:approach	35	14701826.65	420052.19	0.00
Residuals	4,134	10109829.79	2445.53	

Table 5.29 Average SBC Violations per Type.

	$\alpha = 1$			TestFocus			RndOrder			OptOrder		
	As	Ag	I	As	Ag	I	As	Ag	I	As	Ag	I
Ant 1.6.2	269.20	7.80	84.50	113.00	1.00	10.00	366.70	12.10	21.95	323.00	12.00	136.00
Ant 1.7.1	732.55	13.40	200.15	135.35	1.00	4.00	910.55	18.80	63.55	860.00	15.00	162.00
ArgoUML 0.14.0	1,530.85	5.50	253.20	818.65	5.00	53.00	1,531.50	6.30	160.05	1,597.00	8.00	207.00
ArgoUML 0.22.0	1,534.05	9.00	188.75	999.15	10.00	42.00	1,583.20	7.35	134.95	1,833.00	14.00	214.00
Xerces 2.0.1	156.70	0.00	7.85	69.00	0.00	2.00	123.50	0.00	1.25	122.00	0.00	11.00
Xerces 2.6.2	70.85	0.00	9.45	56.00	0.00	2.00	91.80	0.00	3.85	105.00	0.00	14.00

166 to 471 for the optimal order, while it ranges between 22 and 28 for MITER orders with $\alpha = 0$, representing an increase of at least 70%. Similar results are also observed in the other releases. The low performance of random orders in terms of early defect detection and their high number of SBC violations confirm that testers should avoid random orders.

Besides the fact that, we cannot build in practice optimal orders as defect locations are not known, optimal order are impracticable regrading the number of violated SBC. Moreover, in both optimal and random orders, inheritance and aggregation SBC are always violated (see Table 5.29). Indeed, as explained in Section 5.2, violating SBC principle when a method call is involved (association SBC) may not be as critical as when an aggregation or inheritance is part of the coupling.

MITER results with prioritization ($0 < \alpha <= 1$):. Surprisingly, when $\alpha = 0.1$, the number of SBC violations is always very close (or equal) to that of $\alpha = 0$. The slight increase in the number of SBC violations is highly counterbalanced by an improvement in terms of early defect detection. Tables 5.20a, 5.20b, and 5.22b show that increasing α to 0.1 causes an increase less than 10% of numbers of SBC violations but a NEDDR more than doubled (Tables 5.7a, 5.7b and 5.9b). There are orders promoting early defect detection with the same (or similar) number of SBC violations than those produced by traditional CITO approach (MITER without prioritization). However, it is difficult or impossible to find these

orders when the search is only guided by the minimization of SBC violations, *i.e.*, when $\alpha = 0$. Thus, once class prioritization is accounted for with a relatively low weight, the search will continue to be mainly guided by the minimization of SBC violations; however all classes with no dependency among them will be ranked according to their contribution in the maximization of early defect detection, increasing the early defect detection rate.

The number of SBC violations grows with α in all the analyzed releases except in Xerces releases, for which there is almost no variation in the number of SBC violations (cf. Table 5.21) when α varies from 0.1 to 0.75. This increase of the number SBC violations confirms the contribution of class prioritization in the overall search: more α grows, more MITER violates SBC principle to satisfy class prioritization. MITER behavior on Xerces releases is impacted by two factors. First, the defect predictor has a better quality. Second, Xerces is small regarding the number of classes to test. Moreover, the density of relationships among classes of Xerces is less strong than that of the other programs.

For α less or equal to 0.75, MITER always finds solutions with no SBC violations involving aggregation or inheritance except in ArgoUML, where some solutions contain one violated inheritance or aggregation. However, when $\alpha = 1$, *i.e.*, when the search is only guided by class prioritization, in all programs, the number of violations is very high and close to the number of violations in optimal orders. Finally, when $\alpha = 1$, all solutions found by MITER contain SBC violations involving aggregation and inheritance (see Table 5.29); these orders should be avoided.

Comparison between MITER and test focus: As shown in Tables 5.19–5.21, the number of SBC violations for test focus—in all programs, for all coverage thresholds, and for α ranging from 0.1 to 0.75—is significantly greater than that obtained with MITER. The number of SBC violations for test focus is at least twice as big than that obtained with MITER. For example, in ArgoUML 0.22.0, the number of SBC violations is between 393 and 1,052 for test focus while it is between 89 and 153 for MITER. The Mann-Whitney and the Cliff's d tests show that the differences are all significant with a large magnitude except in one program where the magnitude is medium (cf. Tables 5.22–5.27). Moreover, in all the analyzed programs, the violated SBC in test focus orders involve inheritances and–or aggregations. All dependencies between classes in the test focus set and other classes are systematically violated no matter the type of dependency resulting into a high number of (possibly complex) SBC violations.

When $\alpha = 1$, the test focus has significantly smaller number of violations (with a large effect size) because in that situation, the goal of minimizing SBC violations is discarded and the search is only driven by the maximization of early defect detection.

RQ2 Summary: *When both early defect detection and SBC violations are taken into account ($0 < \alpha < 1$), MITER yields to a number of SBC violations significantly smaller than test focus. If the SBC objective is discarded ($\alpha = 1$), test focus performs significantly better.*

Overall Summary: Based on the results of the two RQs, we can answer positively our main research question: *Can MITER provide balanced class integration orders that minimize server-before-client violations and maximize early defect detection capability?* The region where we can find a compromise is the one where $\alpha < 0.5$. Indeed, in general, when α moves slightly away from zero, the defect detection rate considerably increases while the number of SBC violations remains the same or close to the lower bound found for $\alpha = 0$. We thus conclude that MITER can be useful to support testers in finding a good compromise between early defect detection capability and the minimization of SBC violations.

5.6 Threats to Validity

This section discusses the threats to the validity of our study.

To ensure the *construct validity* of our study, we did our best to make sure only reliable information was used when evaluating MITER, however, there are several points that can be further discussed.

A concern for our study is *defect attribution i.e.*, how we assign a defect to a release. As explained in Section 5.4.4, we rely on data from a previous work (Bavota *et al.*, 2012) based on the *SZZ* algorithm (Sliwerski *et al.*, 2005; Kim *et al.*, 2006). *SZZ* is not guaranteed to be 100% accurate, therefore a bug can be attributed to a wrong release. However, it is beyond the scope of this study to have a 100% accurate set of release-related defects. Instead, the goal is to show the performances of MITER in finding orders that promote early defect detection.

A different concern is related to the set of classes to be tested and thus the elements included in $\mathcal{C}$. To this aim we relied on changes and dependencies identified respectively by *ChangeDistiller* (Fluri *et al.*, 2007) and the *Ptidej* suite [8]. However, we cannot be sure that all valid changed and impacted classes are included in $\mathcal{C}$. Also we selected as coverage criterion of the integration testing, the call sites coverage criterion which is the weakest integration test criterion (Lin et Offutt, 1998). We measured this coverage as the number of call sites encountered in each class. We cannot exclude that counting caller–callee relations or using a different coverage criterion will produce different results.

The aim of MITER is to determine a test order, and not to impose any testing criteria nor to generate test data. Therefore, while in our study, the rate of defect detection is an upper

8. http://www.ptidej.net

bound to the defects that can be found. The actual defect detection rate depends on the test strategy followed and the actual test data generated.

The *internal validity* of our study is threatened by the intrinsic randomness of MA. We dealt with that by running the algorithms 20 times and reporting descriptive statistics and–or performing statistical tests where appropriate. Another internal validity threat is related to the particular choice of the analyzed releases. We tried to mitigate it by performing our study using different programs and different releases.

In order to mitigate the threats to *conclusion validity*, wherever appropriate, we used statistical procedures and effect size measures to support our claims. Specifically, we use Mann-Whitney test to check the presence of significant differences between MITER (under various values of α and coverage thresholds) and the test focus approach (Borner et Paech, 2009a). Also, we complement statistical tests with non-parametric effect size measures (Cliff's d) to evaluate the magnitude of the observed differences.

External validity threats to our study are mainly due to the limited number of analyzed releases. Although we expect that similar change impact sets, defect data, types and distribution of dependencies can occur with other programs and thus similar results can be found, further studies need to be conducted to verify such a conjecture.

5.7 Conclusion

This chapter proposes MITER (Minimizing Integration Testing EffoRt), a formalization of the class integration test order problem and an approach. MITER aims at (1) maximizing early defect detection *i.e.*, test with high priority classes having a high (estimated) defect-proneness, such as AP classes, and (2) minimizing the number or overall complexity of violations of the server-before-client principle. MITER uses class test priorities, assigned by experts or computed by defect prediction tools, to define a class testing order promoting early defect detection while minimizing the cost of SBC violations.

MITER uses a memetic algorithm, *i.e.*, a meta-heuristic optimization approach, to calculate a test order based on the new model. We empirically analyzed MITER's ability to produce test orders that promote early defect detection while keeping low the number of SBC violations on multiple releases of three Java programs: Ant, ArgoUML, and Xerces. We compared MITER orders against an optimal order (upper bound), a random ordering (lower bound), and a Boolean classification order based on the approach of Borner et Paech (2009a). The results of the study allowed us to positively answer our high-level research question:

Can MITER provide balanced class integration orders that minimize server-before-client vio-

lations and maximize early defect detection capability?

Because the model underlying MITER integrates both assigned priorities and SBC violations, we found that the higher the importance assigned to the testing priority, the higher the number of SBC violations. However, orders generated with a small value of the testing priority weight can significantly increase the early defect detection rate with no or little increase of SBC violations compared to orders computed without prioritization. Thus, testers can obtain a compromise between these two objectives.

As a general guideline, the value of α should be set low, likely around 0.1. With this value of α, MITER can find orders with the same cost of SBC violations as the ones when $\alpha = 0$ but with a substantial improvement in the early defect detection rate. Then, testers can increase the value of α according to their needs in terms of class prioritization and budget.

We observed that the compromise depends also on the program complexity, the types and the density of relationships among classes, and the quality of the defect prediction.

We also showed that the fine-grained prioritization used in MITER leads to better balanced orders than a Boolean classification as proposed by Borner et Paech (2009a).

We summarize the main outcomes of our empirical study as follows:
- Balanced orders between early defect detection and SBC violations exist and MITER can find them using values of testing priority weight, α, smaller than 0.5.
- In agreement with the general consensus, random orders are not suitable to perform cost-effective class integration testing activities: such orders are not efficient for early defect detection and yield to high numbers of SBC violations including aggregations and inheritances.
- Orders with extreme values of α, *i.e.*, $\alpha = 0$ and $\alpha = 1$, should be avoided because they do not produce a compromise between early defect detection and SBC violations.
- Fine-grained prioritization used in MITER is better in promoting balanced orders between early defect detection and SBC violations than the Boolean classification-based prioritization (Borner et Paech, 2009b).

MITER can then help prioritize defect-prone classes in general and AP classes in particular without a high extra cost regarding the SBC principle. Thus, it can make the test of AP classes more cost-effective.

In the next chapter, along with our goal to aid to a better test of programs containing AP classes, we analyze the usability of Madum testing. We also propose means to improve and reduce the cost of using this testing strategy.

CHAPTER 6 IMPROVING THE USABILITY OF MADUM

In the previous chapter, we proposed MITER, an approach to solve the problem of class integration test order. MITER prioritizes defect-prone classes, such as AP classes, while minimizing the cost of SBC violations. It contributes to our goal of improving the test of OO programs containing APs by making the test of AP classes cost-effective and increasing early defect detection capability.

As shown in chapter 4, testing classes involving APs is expensive but necessary. Indeed, AP classes are more defect-prone than other classes. The present chapter aims at analyzing and improving the usability of Madum testing, a specific object-oriented unit testing. Improving the usability of Madum testing could help reduce testing cost of OO programs containing APs.

State-based testing is a model-based testing strategy because we must provide the state chart of a class to derive sequences of methods to test. Pre-and-post conditions testing requires also defining pre-and-post conditions for each method before identifying sequences of methods to test. However, state charts and pre-and-post conditions are hard to derive automatically because they are based on the semantic of the program. This fact makes the automation of these testing strategies difficult. In contrast to these techniques, Madum relies only on source code analyzis to identify sequences of methods to test. Madum testing appears then as a good candidate for the automation which is one of the main means to reduce testing cost and increase testing use and efficiency (Ammann et Offutt, 2008).

Although Madum testing is a good candidate for automation, the authors of this strategy did not provide formal coverage criteria neither any strategy to generate test data to exercise the identified sequences. Formal criteria are important for an adequate use of the testing strategy and its automation. Moreover, if this testing strategy sounds theoretically useful, we do not have any evidence of its efficiency or usability in practice. Another concern is related to the number of transformers that can highly increase the cost of using Madum testing.

In this chapter, we first propose refactoring actions to reduce the number of transformers and thus the cost of using Madum testing. We define formal criteria to guide in generating Madum test data and help for its automation. Our focus is only on Madum sequences as for the other steps in Madum testing, traditional testing strategies and criteria can be used. Finally, to help automate Madum testing, we formulate the problem of generating test data for Madum as a search-based problem.

6.1 Refactoring for Reducing Madum Testing Cost

Table 6.1 Impact of the reduction of the number of transformers (TRS) on the number of test cases (TC).

Class (program)	Type	Before refactoring TRS	Before refactoring TCs	After refactoring TRS	After refactoring TCs
TokenFilter (Ant)	CDSBP	5	263	2	27
PropPanel (ArgoUML)	Blob	5	271	3	43
BooleanExpressionComplexityCheck (Checkstyle)	LPL	6	732	5	132
AxisState (JFreeChart)	NAP	5	248	1	11
DynamicTimeSeriesCollection (JFreeChart)	Blob	4	208	2	122

According to Madum testing described in 2.1.3, the number of transformers and the number of constructors in each data slice can dramatically increase the number of test cases. Table 6.1 reports data about cases in which specific refactoring activities, discussed below, can be used to reduce the number of transformers in the data slices of such classes. These classes are from the programs used in the experiment of Chapter 4. The proposed refactoring activities do not remove APs, *i.e.*, they are complementary or sometimes in opposition with "traditional" refactorings that one often performs with the aim of increasing comprehensibility and maintainability.

A typical situation that we found in most of the classes reported in Table 6.1 is related to source code fragments that transform the same fields cloned in multiple methods. These cloned statements contribute to increase the number of transformers per slice and consequently the number of test cases. We reduce testing cost by performing an extract method refactoring.

For example, in the class *PropPanel* of ArgoUML, we found a sequence of statements that transforms the field *listenerList* and that is repeated in four methods with a slight variation. These repeated sequences increase the number of transformers for the slice of *listenerList* field and, consequently, the number of test cases required to test that class according to Madum testing. We extract those statements and create a new method that is then called by the old ones. With this refactoring action, the number of transformers for the field *listenerList* becomes 3 instead of 5 and the number of test cases of the new class is 43 instead of 271. This refactoring reduces the number of transformers of this class and then the number of test cases required to test it according to Madum testing.

Another case concerns multiple methods having a very similar code structure and behavior

while having a different name. This is possibly meant to make the source code easier to understand. Those methods transform the same field(s). An example of this case is found in the class *AxisState* of JFreeChart. In this class, we have four methods, namely *cursorUp*, *cursorDown*, *cursorLeft*, and *cursorRight* that increment (*cursorDown* and *cursorRight*) or decrement (*cursorUp* and *cursorLeft*) the field *cursor* by a given value passed as argument. We can reduce the number of test cases by replacing the four methods by a new one, namely *moveCursor*. Then, we replace the call of the old methods by the new one and adjust the argument: a positive argument is passed instead of calling *cursorDown* or *cursorRight* and a negative argument is passed instead of calling *cursorUp* and *cursorLeft*. This refactoring helps to reduce the number of transformers from 5 to 1 and, consequently, the number of test cases. However, this refactoring could negatively affect code understandability. Indeed, the old methods had more appropriate and straightforward names than the new one. As an alternative, it is possible to use the old methods as simple wrappers directly calling the new method *moveCursor*. Thus, the old methods are still used. The code is actually refactored into *moveCursor* and only *moveCursor* must be tested achieving thereby the double goal of not affecting understandability while reducing the number of test cases. This example shows that refactoring performed for the sake of reducing testing cost must be carefully chosen to avoid decreasing other quality attributes, such as understandability.

In summary, the examples reported in Table 6.1 show that there are opportunities for refactoring that can reduce testing cost. All classes reported in our examples except the class AxisState participate in APs and thus their test require a high number of test cases. We suggest that APs refactoring should not only consider actions to improve cohesion, reduce coupling, and in general address all maintainability issues. It should also consider specific refactoring activities, as those described above, aimed at reducing the number of transformers per data slice and thus the number of test cases. Such refactorings can be worthwhile also for some classes that do not participate in APs and that, however, have a high number of transformers, *e.g.*, class *AxisState* of JFreeChart. The examples above show also that, when applying refactoring actions to classes with the purpose of reducing testing costs, we must be aware of the possible impact on other quality attributes and find the best trade-off, *e.g.*, between testability and comprehensibility/maintainability.

6.2 Madum Sequences Coverage Criteria Test

To tackle the problem of sequences of methods to test within a class, Madum testing proposes to test only sequences of transformers. We call these sequences of transformers thereafter *Madum sequences*. The rational behind this strategy is that only transformers change the

value of attributes and therefore the state of the object under test. Thus, only a call to a transformer can put the object in a state that contradicts its axioms. The goal of focusing only on the test of sequences of transformers is to reduce the total number of possible sequences to test without undermining the effectiveness of the test. However, Madum testing does not define the way to generate test data. It is the task of a tester to find meaningful inputs for the different Madum sequences. To make the test of a Madum sequence meaningful, inputs should trigger the execution of the test through the statements that modify the attribute of the slice under testing. Testing a Madum sequence means then finding an execution path that will traverse, in each transformer of a sequence, at least one statement that modifies the attribute. We call these statements *slice attribute transforming statements (SATS)*. In the following, we describe the different coverage criteria that we propose to test Madum sequences based on SATS.

6.2.1 Notation

Let first define some notations used to formalize the criteria that we propose.

We denote a slice by $S_{a_i} =< a_i, M_{a_i} >$ where a_i is the attribute of the slice and M_{a_i} the set of methods that access a_i. M_{a_i} in turn consists of four subsets: C_{a_i} the set of constructors that initialize a_i, R_{a_i} the set of reporters of a_i, T_{a_i} the set of transformers of a_i, and O_{a_i} the set of other methods that access a_i. A Madum sequence corresponds then to a sequel of methods in T_{a_i} appended to one constructor from C_{a_i}. For $C_{a_i} = \{c_1, c_2, \ldots, c_p\}$ and $T_{a_i} = \{t_1, t_2, ..., t_n\}$, the set of Madum sequences for S_{a_i} and denoted SEQ_{a_i} is as follows:

$SEQ_{a_i} = \{seq =< c_j, t_k, t_f, \ldots, t_l > \mid j \in [\![1; p]\!] \text{ and } k, f, \ldots, l \in [\![1; n]\!]\}$.

Let The k_{th} SATS be denoted by Δ^c_{ik} in the constructor c_i and by Δ^t_{ik} in the transformer t_i.

6.2.2 Simple Madum Sequence Coverage (SMSC)

The first Madum sequences coverage criterion that we propose is the simplest one and is defined as follows:

Simple Madum Sequence Coverage (SMSC): the test requirements set TR_{SMSC} contains at least one sequence of slice attribute transforming statements for each Madum sequence of each slice.

More formally,

$\forall X \in SEQ_{a_i} : X =< c_j, t_k, t_f, \ldots, t_l >, \exists\ tr \in TR_{SMSC} \mid tr =< \Delta^c_{je}, \Delta^t_{kq}, \Delta^t_{fs}, \ldots, \Delta^t_{lv} >$.

The SMSC requires that at least one sequence of SATS be covered for each Madum sequence

of each slice. The minimal number of sequences of SATS to cover if we assume that all sequences are feasible is then the total number of Madum sequences.

6.2.3 Complete Madum Sequence Coverage (CMSC)

A transformer may contain more than one SATS and those SATS can be in different paths of execution. As suggested by Bashir et Goel (2000), each of these paths should be exercised during the test to cover all SATS. We thus define the second Madum Sequence coverage criterion as follows:

Complete Madum Sequence Coverage (CMSC): The set of test requirements TR_{CMSC} contains all the sequences of SATS of each Madum sequence. CMSC is formalized as follows:

$\forall X \in SEQ_{a_i} : X =< c_j, t_k, t_f, \ldots, t_l >,\ \forall e \in [\![1; N_e]\!], q \in [\![1; N_q]\!], s \in [\![1; N_s]\!], \ldots,\ and\ v \in [\![1; N_v]\!], \exists\ tr \in TR_{SMSC} \mid tr =< \Delta^c_{je}, \Delta^t_{kq}, \Delta^t_{fs}, \ldots, \Delta^t_{lv} >,$
with $N_e,\ N_q,\ N_s, \ldots,$ *and* N_v *the number of SATS in* $c_j, t_k, t_f, \ldots,$ *and* t_l.

The CMSC requires that all sequences of SATS be covered for each Madum sequence of each slice.

When in a bloc, there are more than one statement that modify the same attribute, we take into consideration only the first one as the other will be executed as soon as the first one is.

6.2.4 Intermediate Madum Sequence Coverage (IMSC)

Because CMSC can require a big set of test cases in presence of many transformers and many SATS per transformer located in different execution paths, we propose an intermediate coverage criterion defined as follows.

Intermediate Madum Sequence Coverage (IMSC): the test requirements set TR_{IMSC} satisfies SMSC and contains all SATS of all transformers of each slice at least once. More formally,

$\forall X \in SEQ_{a_i} : X =< c_j, t_k, t_f, \ldots, t_l >, TR_{IMSC} \Rightarrow TR_{SMSC}\ \&\ \forall \Delta^{c|t}_{ki}, \exists\, tr \in TR_{IMSC} \mid \Delta^{c|t}_{ki} \in tr.$

6.2.5 All Madum Paths Coverage (AMPC)

In the three proposed criteria, there is no constraints on the path between the different SATS; therefore these criteria can then be categorized as node–node-oriented criteria (McMinn, 2004). The test requirement is then a sequence of nodes, each node representing a SATS from different transformers. However, more than one subpath can traverse a given sequence

of SATS and each of them can impact the output. We define another coverage criterion that integrates this aspect.

All Madum Paths Coverage (AMPC): For each sequence of SATS, the test requirements set TR contains all simple paths that traverse those SATS.

Thus, AMPC will require at least as many test cases as CMSC.

6.2.6 Subsumption Relations among Madum Sequences Coverage Criteria and Infeasibility

From the definition of each coverage criterion, we can deduce the following subsumption relations among Madum sequences coverage criteria:

(1) TR_{IMSC} *subsumes* TR_{SMSC};

(2) TR_{CMSC} *subsumes* TR_{IMSC};

(3) TR_{AMPC} *subsumes* TR_{CMSC}.

In summary, SMSC is the weakest coverage criterion whereas AMPC is the strongest one. AMPC subsumes CMSC that in turn subsumes IMSC.

Some sequences of SATS or even some Madum sequences may not be feasible. As the problem of detecting infeasible paths is undecidable, we will consider the set of test requirements of each criterion as an upper bound of possible sequences fo SATS to test.

6.2.7 Example

This section illustrates the proposed coverage criteria through an example. Let S_{a_2} be a slice under test. S_{a_2} contains one constructor ($C_{a_2} = \{c_1\}$) and three transformers ($T_{a_2} = \{t_1, t_2, t_3\}$). The set of Madum sequences of S_{a_2} is then:

$SEQ_{a_2} = \{seq_1 =< c_1, t_1, t_2, t_3 >, seq_2 =< c_1, t_1, t_3, t_2 >, seq_3 =< c_1, t_2, t_1, t_3 >, seq_4 =< c_1, t_2, t_3, t_1 >, seq_5 =< c_1, t_3, t_1, t_2 >, seq_6 =< c_1, t_3, t_2, t_1 >\}$

Let us further assume the following statements:
– c_1 contains one SATS (Δ^c_{11});
– t_1 contains two SATS ($\Delta^t_{11}, \Delta^t_{12}$) in different paths of execution;
– t_2 contains three SATS ($\Delta^t_{21}, \Delta^t_{22}, \Delta^t_{23}$) in different paths of execution;
– t_3 contains two SATS ($\Delta^t_{31}, \Delta^t_{32}$) in different paths of execution.
Figure 6.1 depicts a graph that represents all sequences of SATS for the first sequence of S_{a_2}: $seq_1 =< c_1, t_1, t_2, t_3 >$. Each node in the graph corresponds to a SATS and each path to a

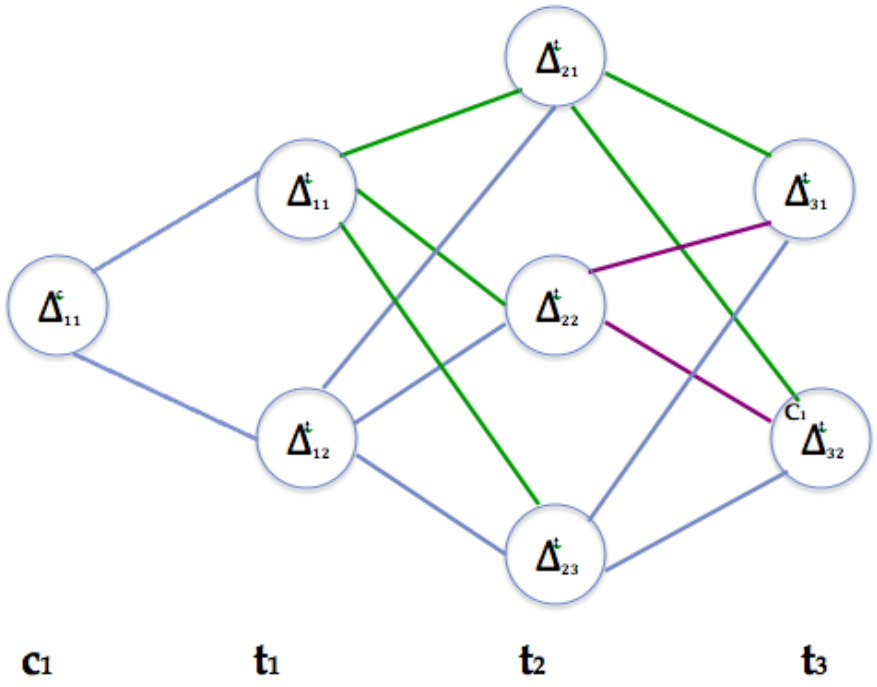


Figure 6.1 Graph representing sequences of SATS in a Madum sequence.

sequence of SATS. This graph helps derive the different sets of test requirements for each criterion for this specific sequence. Another sequence will lead to another graph.

SMSC requires that at least one SATS of each transformer including the constructor be covered. Thus, each path of the graph satisfies this criterion. For example, the path $p1 =< \Delta^c_{11}, \Delta^t_{11}, \Delta^t_{21}, \Delta^t_{31} >$ fulfils the requirement as one SATS of the constructor and each transformer is traversed by the path.

To satisfy IMSC, the set of tests should satisfy SMSC but also include all SATS of seq_1. Because of the second condition of this requirement, there is no single path that satisfies this criterion. The minimum number of paths that fulfils the requirement is equal to the maximum SATS contained in a transformer. In this example, the criterion will then require at least three paths. An example of set of paths that fulfils the requirement is the set $\{p1, p2, p3\}$ with $p_1 =< \Delta^c_{11}, \Delta^t_{11}, \Delta^t_{21}, \Delta^t_{31} >$, $p_2 =< \Delta^c_{11}, \Delta^t_{12}, \Delta^t_{22}, \Delta^t_{31} >$, and $p_3 =< \Delta^c_{11}, \Delta^t_{11}, \Delta^t_{23}, \Delta^t_{32} >$.

To satisfy CMSC, one should exert all paths of the graph as the criterion requires that all sequences of SATS to be covered. Thus, the number of test cases for the CMSC is 12.

For the last criterion (AMPC), all possible paths of the code that traverse all paths of the graph should be covered.

The rest of this chapter will deal with the three first criteria; the last one will be considered in future research.

6.3 Towards Madum Testing Automation

In this section, we take the first steps to automate Madum testing based on the coverage criteria that we proposed in the previous section. Search-based techniques have been proved to be efficient in test data generation (McMinn, 2004). Consequently, we formulate the problem of generating Madum test data as a search-based problem by defining in the following the two key elements of such a formulation: the representation of the problem and the fitness function (Harman *et al.*, 2012).

6.3.1 Problem Formulation

Testing Madum sequences consists in covering at least one sequence of SATS or at most all sequences of SATS. The goal is then to find a set of parameters that triggers the execution of the Madum sequence through the paths that traverse the SATS of those sequences.

In most of the search-based techniques proposed to generate OO unit test data, white box testing in particular (Tonella, 2004; Fraser et Arcuri, 2013; Panichella *et al.*, 2015), a candidate solution is either a test case (Tonella, 2004; Panichella *et al.*, 2015) or a test suite (Fraser et Arcuri, 2013). A test case is defined as a sequence of statements of a certain length and a test suite as a set of test cases (Fraser et Arcuri, 2013). Things are slightly different when generating test data for Madum sequences. Indeed, the sequence of methods to call on the object under test is known. What is unknown is the arguments to invoke the constructor and the transformers of the sequence. For the sake of simplicity, we consider a candidate solution as an input vector of objects and–or primitives that map the parameters of the methods in the sequence. The final test case will consist of statements required to create the different objects, the calls to the constructor and the methods in the Madum sequence, and assert statements to check the state of the object under test.

Let seq_i be the Madum sequence to test and tr_i the sequence of SATS of seq_i to cover. $seq_i = \{c_j(P_{c_j}), t_k(P_{t_k}), t_f(P_{t_f}), \ldots, t_l(P_{t_l})\}$, where $P_m = (p_{m_1}, p_{m_2}, \ldots, p_{m_n})$ represents the list of parameters of method m.

A configuration (potential solution) of the search problem is any vector

$V = \{V_{c_j}, V_{t_k}, V_{t_f}, \ldots, V_{t_l}\}$ where $V_m = (v_{m_1}, v_{m_2}, \ldots, v_{m_n})$ is a list of arguments to invoke method m.

V will be a solution of the search if and only if the execution path of the sequence of transformers seq_i when using arguments in V contains the sequence of SATS tr_i.

6.3.2 Fitness Function

As mentioned in Section 6.2.5, the three first Madum sequences coverage criteria are node-to-node oriented criteria. Fitness function for such criteria is a cumulative node-oriented strategy (McMinn, 2004). Thus, as mentioned in (McMinn, 2004), the fitness value of a given configuration to cover a sequence of SATS is a cumulative of fitness values of that configuration to reach each SATS.

Let $tr_i =< \Delta^c_{je}, \Delta^t_{kq}, \Delta^t_{fs}, \ldots, \Delta^t_{lv} >$ be the target of the search.

The fitness function that evaluates the cost to reach the SATS Δ_s in tr_i corresponds to the node oriented fitness function defined as:

$$F(\Delta_s) = level\_distance(\Delta_s) + Norm(branch\_distance(\Delta_s)) \tag{6.1}$$

where $level\_distance(\Delta_s)$ is the approach level of the node, $branch\_distance(\Delta_s)$ its branch distance, and $Norm$ a normalization function that prevents the $branch\_distance$ value dominating the $level\_distance$ value. $F(\Delta_s) = 0$ indicates that the statement has been reached.

The approach level is a metric that describes the distance (number of branching nodes) between the branching node of interest and the branching node where the execution diverges. The branch distance is a metric that indicates how close the predicate of the analyzed branching node is to being true (McMinn, 2004).

The fitness function that will guide the search towards the target tr_i is defined as:

$$F(S_i) = \sum_{i=1}^{p} Norm(F(\Delta_s)) \tag{6.2}$$

where $F(\Delta_s)$ is the fitness value of the SATS Δ_s and $Norm$ a normalization function that prevents the fitness value of one of the SATS dominating the others.

6.4 Conclusion

This chapter analyzed and proposed means to improve the usability of Madum testing. We first showed on a set of classes how specific refactoring actions could possibly contribute to reduce the cost of testing classes in general and classes involving APs in particular when

using Madum testing. These kinds of refactoring are different, complementary, and in some cases can pursue conflictual objectives with respect to traditional refactoring actions aimed at improving comprehensibility and maintainability.

We also proposed formal coverage criteria to facilitate the use of Madum testing and guide in generating test data.

Finally, we formulated the problem of generating test data for Madum as a search-based problem in accordance with the proposed coverage criteria. This formulation is the first step towards the automation of Madum testing.

CHAPTER 7 CONCLUSION

This chapter summarizes the results and conclusions of our thesis. It also presents possible future directions.

7.1 Contributions

Our modern society is highly computerized. Therefore, testing is paramount of importance. Although expensive, software testing remains the primary means to ensure software dependability. Unfortunately, the main features of object-oriented (OO) paradigm—one of the most popular development paradigms—complicate testing activities. Our thesis statement was: *The cost of testing classes involving antipatterns is higher than that of other classes but it is possible to offer techniques to reduce that cost during unit and integration testing.* This thesis is a contribution to the global effort of researchers over the two past decades to reduce OO programs testing cost. The study of factors that could impair OO programs testing is one of the main investigated research directions. Indeed, the knowledge of factors that impede testing cost or its effectiveness could help to propose adequate solutions and approaches. However, to the best of our knowledge, our study is the first study regarding the potential negative impact of antipatterns on OO testing.

Antipatterns are defined as recurring and poor design or implementation choices (Brown *et al.*, 1998). Past and recent studies show that antipatterns negatively impact many software quality attributes, such as maintenability (Deligiannis *et al.*, 2003) and understandability (Abbes *et al.*, 2011). Other studies also report their low resilience to change and defect (Olbrich *et al.*, 2009; Khomh *et al.*, 2012). We performed an empirical study to assess the impact of antipatterns on the cost of unit testing of OO programs. Using DECOR (Moha *et al.*, 2010), we identified AP occurrences in four open-source java programs: Ant 1.8.3, ArgoUML 0.20, Checkstyle 4.0, and JFreeChart 1.0.13. We quantified class testing cost using the number of test cases required to test each class with Madum testing (Bashir et Goel, 2000). The results showed that indeed antipatterns negatively impact class testing cost: AP classes are in general more expensive to test than other classes. This impact varies depending on the kind of APs: classes involving some kinds of APs, such as Blob, AntiSingleton or ComplexClass, require a higher number of test cases whereas some other APs, such as MethodChains or LazyClass, do not strongly contribute to the class testing cost. In this empirical study, we also analysed the cost-effectiveness of prioritizing the test of AP classes. The results showed that prioritizing the test of APs may be worthwhile. Although

expensive, the test of APs may allow detecting most of the defects and early.

Then, we proposed MITER (Minimizing Integration Testing EffoRt), a new formalization and approach to the class integration test order (CITO) problem. The CITO problem is one of the major problems when integrating classes in OO programs. This problem concerns the minimization of the cost related to the order in which classes should be integrated and tested. This cost is usually measured in terms of violations of the server-before-client principle (*i.e.*, stubs). Contrary to most of existing approaches to solve the CITO problem, MITER aims to minimize SBC violations and maximize early defect detection. Indeed, as shown in our first contribution, prioritizing defect-prone classes, such as AP classes, could increase early defect detection. Thus, MITER proposes to test first classes having a high (estimated) defect-proneness, such as AP classes.

MITER relies on experts or defect prediction tools to assign test priorities to classes. MITER uses a memetic algorithm, *i.e.*, a meta-heuristic optimization approach, to generate an integration test order promoting early defect detection while minimizing the number of SBC violations. We conducted an empirical study to evaluate the ability of MITER to produce balanced test orders between the two objectives. We used in this study multiple releases of three Java programs: Ant, ArgoUML, and Xerces. We compared MITER orders against an optimal order (upper bound), a random ordering (lower bound), and a Boolean classification approach(Borner et Paech, 2009a). Results of the study report that: (i) MITER outperforms random ordering, (ii) MITER can generate balanced test orders between SBC violations and early defect detection, and (iii) the fine-grained class prioritization used by MITER is better than a Boolean classification. MITER can then help prioritizing defect-prone classes in general and AP classes in particular without a high extra cost regarding the SBC principle. It can thus be useful to support testers in finding a trade-off between early defect detection capability and the minimization of SBC violations.

As shown in our first contribution, AP classes are expensive to test but because they are more defect-prone (Khomh *et al.*, 2012), they must be thoroughly tested. In our third contribution, we analyzed and improved the usability of Madum testing, a specific OO unit testing strategy to help in the test of AP classes. Madum testing is one of the testing strategies proposed to overcome the limitations of traditional unit testing strategies in the test of OO programs. The main advantage of Madum testing compared to other OO unit testing strategies—state-based testing and pre-and-post conditions—is that it does not require other specific documentation to identify test cases; it relies only on the source code.

Madum testing is then a good candidate for automation which is one of the best ways to reduce testing cost and increase testing reliability (Ammann et Offutt, 2008). Moreover, the

analysis of the fundamentals of Madum testing shows that the number of methods of a class that modify a given attribute (transformers) is a key factor in the cost of the test.

Therefore, we proposed specific refactoring actions to reduce this cost and thus reduce the cost of testing classes in general and AP classes in particular. Indeed, most of the classes with a high number of transformers involve APs. Applying these refactorings on a set of classes showed that they reduce the number of transformers and thus testing cost. The proposed refactoring are different, complementary, and in some circumstances, can pursue conflictual objectives with respect to traditional refactoring actions aimed at improving understandability and maintainability. To address the lack of specific coverage criteria for Madum testing, we proposed formal coverage criteria to guide in generating test data and automating Madum testing. Finally, we formulated the problem of generating test data for Madum testing as a search-based problem according to the proposed coverage criteria.

The results of our contributions prove our thesis: we bring evidence that APs negatively impact OO testing cost but because of their defect-proneness, focusing on their test could be cost-effective. Using this evidence, we propose MITER to improve class integration testing by providing class integration test orders that minimize SBC violations and increase early defect detection capability. Because AP classes are expensive to test but need to be tested, we proposed refactoring actions to reduce the cost of using Madum testing, a specific OO unit testing. We also define formal coverage criteria to guide in identifying test data for Madum testing. Finally, we proposed a search-based formulation of the problem of automatically generating test data for Madum testing.

7.2 Future Work

We proved our thesis through the different contributions but we also opened opportunities for future research directions. One of these opportunities is to study the effectiveness of Madum testing. In this thesis, we take the first steps towards the automation of this testing strategy. There is a need to finalize this automation and analyze to what extent Madum testing can complement traditional testing strategies in the test of OO programs.

Another study in that direction concerns the effectiveness of Madum testing compared to other OO unit testing strategies, such as state-based testing and pre-and-post conditions testing. An experiment with subjects that will manually write test cases for the different strategies can help address this question. The strategies can then be compared to each other in terms of effectiveness but also in terms of the effort they require.

It will also be interesting to compare the different Madum testing coverage criteria proposed

in the present dissertation in terms of efficiency.

Replicate our study using more Java programs as well programs developed in other languages could help generalize our findings regarding the impact of APs on OO unit testing. Along the same lines, using different OO unit testing strategies as well as traditional unit testing strategies could also be interesting.

As shown by the refactoring we proposed, a high number of transformers can be a refactoring opportunity to reduce the cost of using Madum testing. Another research direction is to investigate the feasibility of automatic or semi-automatic approaches to perform these kinds of refactoring and promote refactoring driven-testability. It will also be interesting to analyze the impact of such refactoring on the cost of other OO unit testing strategies, such as state-based testing. Indeed, transformers are the methods that can lead to a change of state. Having less transformers can reduce the state chart and thus the cost of state-based testing.

The results of the experiment on the performance of MITER suggest that the quality of the predictors could have an impact on the quality of orders generated by MITER. Future work will verify this conjecture by evaluating the impact of the quality of defect predictors on MITER's performance. Implementing MITER model using multi-objectives optimization techniques could also be an interesting future work. Such techniques will provide the set of possible non-dominant orders: orders maximizing early defect detection and minimizing SBC violations but also orders favoring more or less one of the objectives. Thus, testers could choose in that set the order that meets their needs without having to perform multiple trials or setting the parameters of the current formalization.

REFERENCES

Abbes, Marwen and Khomh, Foutse and Gueheneuc, Yann-Gael and Antoniol, Giuliano (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 181–190.

Aynur Abdurazik and Jeff Offutt (2009). Using coupling-based weights for the class integration and test order problem. *The Computer Journal*, *52*(5), 557–570.

Ali, S. and Briand, L.C. and Hemmati, H. and Panesar-Walawege, R.K. (2010). A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering (TSE)*, *36*(6), 742–762.

Paul Ammann and Jeff Offutt (2008). *Introduction to software testing*. Cambridge University Press.

Antoniol, Giuliano and Ayari, Kamel and Di Penta, Massimiliano and Khomh, Foutse and Guéhéneuc, Yann-Gaël (2008). Is it a bug or an enhancement?: A text-based approach to classify change requests. *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*. 304–318.

Andrea Arcuri and Xin Yao (2007). A memetic algorithm for test data generation of object-oriented software. *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*. 2048–2055.

Arisholm, E. and Briand, L.C. and Fuglerud, Magnus (2007). Data mining techniques for building fault-proneness models in telecom java software. *Proceedings of the International Symposium on Software Reliability (ISSRE)*. 215–224.

Arisholm, Erik and Briand, Lionel C. and Johannessen, Eivind B. (2010). A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software (JSS)*, *83*(1), 2–17.

Assunção, Wesley Klewerton Guez and Colanzi, Thelma Elita and Pozo, Aurora Trinidad Ramirez and Vergilio, Silvia Regina (2011). Establishing integration test orders of classes with several coupling measures. *Proceedings of Genetic and Evolutionary Computation Conference (GECCO)*. 1867–1874.

Assunção, Wesley Klewerton Guez and Colanzi, Thelma Elita and Vergilio, Silvia Regina and Pozo, Aurora (2014). A multi-objective optimization approach for the integration and test order problem. *Information Sciences*, *267*, 119–139.

Bache, Richard and Mullerburg, Monika (1990). Measures of testability as a basis for quality assurance. *Journal of Systems and Software (JSS)*, *5*(2), 86–92.

Adrian Bachmann and Christian Bird and Foyzur Rahman and Premkumar T. Devanbu and Abraham Bernstein (2010). The missing links: bugs and bug-fix commits. *Proceedings of the International Symposium on Foundations of Software Engineering*. 97–106.

Badri, Linda and Badri, Mourad and Toure, Fadel (2010). Exploring empirically the relationship between lack of cohesion and testability in object-oriented systems. *Advances in Software Engineering*, Springer Berlin Heidelberg, vol. 117. 78–92.

Rose D. Baker (1995). Modern permutation test software. E. Edgington, éditeur, *Randomization Tests*. Marcel Decker.

Bashir, Imran and Goel, Amrit L. (2000). *Testing Object-Oriented Software: Life-Cycle Solutions*. Springer-Verlag New York, Inc., première édition.

Bavota, G. and De Carluccio, B. and De Lucia, A. and Di Penta, M. and Oliveto, R. and Strollo, O. (2012). When does a refactoring induce bugs? an empirical study. *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 104–113.

B. Beizer (1990). *Software Testing Techniques 2nd edition*. International Thomson Computer Press.

Binder, Robert V. (1999). *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc.

Boehm, B. W. and Papaccio, P. N. (1988). Understanding and controlling software costs. *IEEE Transactions on Software Engineering (TSE)*, *14*(10), 1462–1477.

Booch, Grady (1991). *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

Borner, L. and Paech, B. (2009a). Integration test order strategies to consider test focus and simulation effort. *Proceedings of the International Conference on Advances in System Testing and Validation Lifecycle (VALID)*. 80–85.

Borner, L. and Paech, B. (2009b). Using dependency information to select the test focus in the integration testing process. *Proceedings of Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*. 135–143.

Bourque, Pierre and Fairley, Richard E. (Dick) (2014). *Guide to the Software Engineering Body of Knowledge, Version 3.0*. IEEE Computer Society.

Bowes, David and Hall, Tracy and Gray, David (2012). Comparing the performance of fault prediction models which report multiple performance measures: Recomputing the confu-

sion matrix. *Proceedings of the International Conference on Predictive Models in Software Engineering (PROMISE)*. 109–118.

Boyapati, Chandrasekhar and Khurshid, Sarfraz and Marinov, Darko (2002). Korat: Automated testing based on java predicates. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 123–133.

L.C. Briand and J. Feng and Y. Labiche (2002a). Experimenting with genetic algorithms to devise optimal integration test orders. Rapport technique SCE-02-03, Software Quality Engineering Laboratory, Department of Systems and Computer Engineering, Carleton University.

Briand, Lionel and Pfahl, Dietmar (1999). Using simulation for assessing the real impact of test coverage on defect coverage. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. 148–157.

Briand, Lionel C. and Feng, Jie and Labiche, Yvan (2002b). Using genetic algorithms and coupling measures to devise optimal integration test orders. *International Conference on Software Engineering & Knowledge Engineering*. ACM, 43–50.

Briand, Lionel C. and Feng, Jie and Labiche, Yvan (2003a). Experimenting with genetic algorithms to devise optimal integration test orders. T. Khoshgoftaar, éditeur, *Software Engineering with Computational Intelligence*, Springer US, vol. 731. 204–234.

Lionel C. Briand and Yvan Labiche and Yihong Wang (2001). Revisiting strategies for ordering class integration testing in the presence of dependency cycles. *International Symposium on Software Reliability Engineering*. IEEE Computer Society, 287–297.

Lionel C. Briand and Yvan Labiche and Yihong Wang (2003b). A comprehensive and systematic methodology for client-server class integration testing. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. 14–25.

Brooks,Jr., Frederick P. (1987). No silver bullet essence and accidents of software engineering. *IEEE Computer*, *20*(4), 10–19.

William J. Brown and Raphael C. Malveau and Hays W. McCormick III and Thomas J. Mowbray (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc.

Bruntink, Magiel and van Deursen, Arie (2006). An empirical study into class testability. *Journal of Systems and Software (JSS)*, *79*(9), 1219–1232.

Chatzigeorgiou, Alexander and Manakos, Anastasios (2010). Investigating the evolution of bad smells in object-oriented code. *Proceedings of the International Conference on the Quality of Information and Communications Technology*. 106–115.

S. R. Chidamber and C. F. Kemerer (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493.

Cook, T. D and Campbell, D. T. (1979). *Quasi-experimentation : design & analysis issues for field settings*. Houghton Mifflin.

da Veiga Cabral, Rafael and Pozo, Aurora and Vergilio, Silvia Regina (2010). A pareto ant colony algorithm applied to the class integration and test order problem. *Testing Software and Systems*, Springer. 16–29.

Dabney, James B. and Barber, Gary and Ohi, Don (2006). Predicting software defect function point ratios using a bayesian belief network. *Proceedings of the PROMISE workshop*.

Dahl, O. J. and Dijkstra, E. W. and Hoare, C. A. R. (1972). *Structured Programming*. Academic Press Ltd.

Marco D'Ambros and Michele Lanza and Romain Robbes (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, *17*(4-5), 531–577.

Deligiannis, Ignatios and Shepperd, Martin and Roumeliotis, Manos and Stamelos, Ioannis (2003). An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software (JSS)*, *65*(2), 127–139.

Deligiannis, Ignatios S. and Stamelos, Ioannis and Angelis, Lefteris and and Roumeliotis, Manos and Shepperd, Martin (2004). A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software (JSS)*, *72*(2), 129 – 143.

Bart Du Bois and Serge Demeyer and Jan Verelst and Tom Mens and Marijn Temmerman (2006). Does god class decomposition affect comprehensibility? *Proceedings of the IASTED International Conference on Software Engineering*. 346–355.

Eades, Peter and Lin, Xuemin and Smyth, W. F. (1993). A fast and effective heuristic for the feedback arc set problem. *Inf. Process. Lett.*, *47*(6), 319–323.

Elbaum, Sebastian and Malishevsky, Alexey G. and Rothermel, Gregg (2000). Prioritizing test cases for regression testing. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 102–112.

Ellims, Michael and Bridges, James and Ince, Darrel C. (2006). The economics of unit testing. *Empirical Software Engineering*, *11*(1), 5–31.

Festa, Paola and Pardalos, PanosM. and Resende, MauricioG.C. (2009). Feedback set problems feedback set problems. *Encyclopedia of Optimization*, Springer US. 1005–1016.

Michael Fischer and Martin Pinzger and Harald Gall (2003). Populating a release history database from version control and bug tracking systems. *Proceedings of IEEE International Conference on Software Maintenance (ICSM)*. 23–32.

Beat Fluri and Michael Würsch and Martin Pinzger and Harald Gall (2007). Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering (TSE)*, *33*(11), 725–743.

Marios Fokaefs and Nikolaos Tsantalis and Eleni Stroulia and Alexander Chatzigeorgiou (2011). JDeodorant: identification and application of extract class refactorings. *Proceedings of the International Conference on Software Engineering (ICSE)*. 1037–1039.

Fowler, Martin (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc.

Gordon Fraser and Andrea Arcuri (2013). Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)*, *39*(2), 276–291.

Gordon Fraser and Andrea Arcuri and Phil McMinn (2015). A memetic algorithm for whole test suite generation. *Journal of Systems and Software (JSS)*, *103*, 311–327.

Glenford, J. Myers and Corey, Sandler and Tom, Badgett and Todd, M. Thomas (2004). *The Art of Software Testing*. John Wiley & Sons.

Goldberg, David E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Pub Co.

Grissom, R.J. and Kim, J.J. (2005). *Effect sizes for research: a broad practical approach*. Lawrence Erlbaum Associates.

Tibor Gyimóthy and Rudolf Ferenc and István Siket (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering (TSE)*, *31*(10), 897–910.

Vu Le Hanh and Kamel Akif and Yves Le Traon and Jean-Marc Jézéquel (2001). Selecting an efficient oo integration testing strategy: An experimental comparison of actual strategies. *Proceedings of the European Conference on Object-Oriented Programming*. 381–401.

Harman, M. and Yue Jia and Yuanyuan Zhang (2015). Achievements, open problems and challenges for search based software testing. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 1–12.

Harman, Mark and Jones, Bryan F. (2001). Search based software engineering. *Information and Software Technology*.

Harman, Mark and McMinn, Phil and de Souza, Jerffeson Teixeira and Yoo, Shin (2012). Search based software engineering: Techniques, taxonomy, tutorial. Springer-Verlag. 1–59.

Mary Jean Harrold and John D. McGregor and Kevin J. Fitzpatrick (1992). Incremental testing of object-oriented class structures. *Proceedings of the International Conference on Software Engineering (ICSE)*. 68–80.

S. Holm (1979). A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal on Statistics*, *6*, 65–70.

Hoos, Holger and Sttzle, Thomas (2004). *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann.

Inozemtseva, Laura and Holmes, Reid (2014). Coverage is not strongly correlated with test suite effectiveness. *Proceedings of the International Conference on Software Engineering (ICSE)*. 435–445.

James, Gareth and Witten, Daniela and Hastie, Trevor and Tibshirani, Robert (2014). *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated.

S. Jungmayr (2002). Identifying test-critical dependencies. *Proceedings of IEEE International Conference on Software Maintenance (ICSM)*, 04–13.

Karp, Richard M. (1972). Reducibility among combinatorial problems. R. E. Miller et J. W. Thatcher, éditeurs, *Complexity of Computer Computations*. Plenum Press, New York, 85–103.

F. Khomh and Massimiliano Di Penta and Yann-Gael Gueheneuc (2009). An exploratory study of the impact of code smells on software change-proneness. *Proceedings of IEEE Working Conference on Reverse Engineering (WCRE)*. 75–84.

Khomh, F. and Di Penta, Massimiliano and Guéhéneuc, Yann-Gaël and Antoniol, Giuliano (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, *17*(3), 243–275.

Khomh, F. and Vaucher, Stephane and Guéhéneuc, Yann-Gaël and Sahraoui, Houari (2011). BDTEX: A GQM-based bayesian approach for the detection of antipatterns. *Journal of Systems and Software (JSS)*, *84*(4), 559–572.

Sunghun Kim and Thomas Zimmermann and Kai Pan and E. James Whitehead Jr. (2006). Automatic identification of bug-introducing changes. *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 81–90.

David Chenho Kung and Jerry Gao and Pei Hsia and Jeremy Lin and Yasufumi Toyoshima (1995). Class firewall, test order, and regression testing of object-oriented programs. *Journal of Object-Oriented Programming*, *8*(2), 51–65.

Zhenyi Lin and A. Jefferson Offutt (1998). Coupling-based criteria for integration testing. *Software Testing, Verification and Reliability*, *8*(3), 133–154.

Lloyd, Errol L. and Malloy, Brian A. (2005). A study of test coverage adequacy in the presence of stubs. *Journal of Object Technology*, *4*, 117–137.

Abdou Maiga and Nasir Ali and Neelesh Bhattacharya and Aminata Sabané and Yann-Gaël Guéhéneuc and Giuliano Antoniol and Esma Aimeur (2012). SMURF: a SVM-based incremental anti-pattern detection approach. *Proceedings of IEEE Working Conference on Reverse Engineering (WCRE)*.

Brian A. Malloy and Peter J. Clarke and Errol L. Lloyd (2003). A parameterized cost model to order classes for class-based testing of c++ applications. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*. 353–364.

Chengying Mao and Yansheng Lu (2005). Aicto: An improved algorithm for planning inter-class test order. *Proceedings of the International Conference on Computer and Information Technology*. 927–931.

Marinescu, Radu (2004). Detection strategies: Metrics-based rules for detecting design flaws. *Proceedings of IEEE International Conference on Software Maintenance (ICSM)*. 350–359.

Phil McMinn (2004). Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, *14*(2), 105–156.

McMinn, Phil and Holcombe, Mike (2005). Evolutionary testing of state-based programs. *Proceedings of Genetic and Evolutionary Computation Conference (GECCO)*. 1013–1020.

Mende, Thilo and Koschke, Rainer (2010). Effort-aware defect prediction models. *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*. 107–116.

Moha, Naouel and Guéhéneuc, Yann-Gaël and Duchien, Laurence and Le Meur, Anne-Francoise (2010). DECOR: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering (TSE)*, *36*, 20–36.

Moscato, Pablo and Cotta, Carlos and Mendes, Alexandre (2004). Memetic algorithms. *New Optimization Techniques in Engineering*, Springer Berlin Heidelberg, vol. 141. 53–85.

Moscato, Pablo and others (1989). On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, *826*.

Olbrich, Steffen and Cruzes, Daniela S. and Basili, Victor and Zazworka, Nico (2009). The evolution and impact of code smells: A case study of two open source systems. *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. 390–400.

Thomas J. Ostrand and Elaine J. Weyuker and Robert M. Bell (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering (TSE)*, *31*(4), 340–355.

Palomba, F. and Bavota, G. and Di Penta, M. and Oliveto, R. and Poshyvanyk, D. and De Lucia, A. (2015). Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering (TSE)*, *41*(5), 462–489.

Annibale Panichella and Fitsum Meshesha Kifetew and Paolo Tonella (2015). Reformulating branch coverage as a many-objective optimization problem. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 1–10.

Peters, Fayola and Menzies, Tim and Marcus, Andrian (2013). Better cross company defect prediction. *Proceedings of the Working Conference on Mining Software Repositories*. 409–418.

Riel, Arthur J. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., première édition.

Romano, Daniele and Raila, Paulius and Pinzger, Martin and Khomh, Foutse (2012). Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. *Proceedings of IEEE Working Conference on Reverse Engineering (WCRE)*. 437–446.

Rothermel, Gregg and Untch, Roland H. and Chu, Chengyun and Harrold, Mary Jean (1999). Test case prioritization: An empirical study. *Proceedings of IEEE International Conference on Software Maintenance (ICSM)*. 179–188.

Abdelilah Sakti and Gilles Pesant and Yann-Gaël Guéhéneuc (2015). Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering (TSE)*, *41*(3), 294–313.

Sheskin, David J. (2007). *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, quatrième édition.

Jacek Sliwerski and Thomas Zimmermann and Andreas Zeller (2005). When do changes induce fixes? *Proceedings of the International Workshop on Mining Software Repositories*.

Gilbert Syswerda (1990). Schedule optimization using genetic algorithms. *Handbook of Genetic Algorithms*, 332–349.

Kuo-Chung Tai and Fonda J. Daniels (1997). Test order for inter-class integration testing of object-oriented software. *Proceedings of the International Computer Software and Applications Conference*. 602–607.

Paolo Tonella (2004). Evolutionary testing of classes. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. 119–128.

Yves Le Traon and Thierry Jéron and Jean-Marc Jézéquel and Pierre Morel (2000). Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability*, *49*(1), 12–25.

Travassos, Guilherme and Shull, Forrest and Fredericks, Michael and Basili, Victor R. (1999). Detecting defects in object-oriented designs: using reading techniques to increase software quality. *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*. 47–56.

Nikolaos Tsantalis and Alexander Chatzigeorgiou (2009). Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering (TSE)*, *35*(3), 347–367.

Vergilio, Silvia Regina and Pozo, Aurora and Árias, João Carlos Garcia and da Veiga Cabral, Rafael and Nobre, Tiago (2012). Multi-objective optimization algorithms applied to the class integration and test order problem. *International Journal on Software Tools for Technology Transfer*, *14*(4), 461–475.

Zhengshan Wang and Bixin Li and Lulu Wang and Qiao Li (2011). A brief survey on automatic integration test order generation. *International Conference on Software Engineering & Knowledge Engineering*. Knowledge Systems Institute Graduate School, 254–257.

Bruce F. Webster (1995). *Pitfalls of Object Oriented Development*. M & T Books, 1^{st} édition.

Wohlin, Claes and Runeson, Per and Höst, Martin and Ohlsson, Magnus C. and Regnell, Bjöorn and Wesslén, Anders (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers.

Yamashita, Aiko and Moonen, Leon (2013). Exploring the impact of inter-smell relations on software maintainability: An empirical study. *Proceedings of the International Conference on Software Engineering (ICSE)*. 682–691.

Yin, Robert K. (2008). *Case Study Research: Design and Methods*. Sage Publications, seconde édition.

Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, *22*(2), 67–120.

Zhu, Hong and Hall, Patrick A. V. and May, John H. R. (1997). Software unit test coverage and adequacy. *ACM Comput. Surv.*, *29*(4), 366–427.

Zimmermann, Thomas and Nagappan, Nachiappan (2008). Predicting defects using network analysis on dependency graphs. *Proceedings of the 30th International Conference on Software Engineering*. ACM, 531–540.

APPENDIX A BankAccount SOURCE CODE

This Appendix presents the source code of the class BankAcoount used to illustrate Madum testing stategy.

Listing A.1 BankAccount Class Source Code.

```
public class BankAccount{

    private static interestRate=0;

    private long accountNumber;
    private final String name;
    private double balance;

    public BankAccount(String aName){
        this(aName,0);
    }

    public BankAccount(String aName,double initialBalance){
        this.accountNumber=
                AccountNumberGenerator.generateAccountNumber();
        this.name=aName;
        this.balance = initialBalance;
    }

    public double getBalance() {
        return this.balance;
    }

    public String getName() {
        return this.name;
    }

    public long getAccountId() {
        return this.name;
```

```
}

public void payInterest() {
    this.balance=this.balance*(1+interestRate);
}

public boolean deposit(double amount){

    boolean depositSuccess=false;

    if ( amount >= 0){
        depositSuccess=true;
        this.balance += amount;
    }else{
        System.out.println("Your deposit has to be >= 0");
    }
    return depositSuccess;
}

public boolean withdraw(double amount){

    boolean withdrawSuccess=false;

    if (amount <= 0){
        System.out.println("Withdrawal must be > 0");
    }else if (amount > this.balance){

         System.out.println("Insufficient funds!");

    }else{

        this.balance-=amount;
        withdrawSuccess=true;
    }
    return withdrawSuccess;
}
```

```
    public void printAccount(){

        String description="Account: "this.accountNumber+"\n";
        description="Account Owner: "this.name+"\n";
        description="Account Balance: "this.balance"\n";
        System.out.println(description);

    }

    public static double getInterestRate() {
        return this.interestRate;
    }

    public static void changeInterestRate(double newRate) {
        BankAccount.interestRate = newRate;
    }

}
```

APPENDIX B SOLVING THE CITO PROBLEM: EFFECTIVENESS OF MA AND GA

Briand *et al.* (2002b) presented a genetic algorithm (GA) to solve the class integration test order (CITO) problem. The results of the experiment show that the proposed genetic algorithm performs at least as better as the existing graph-based approaches. The size of the systems used in the experiment goes from 19 classes to 61 classes. The authors justify the use of such small systems by the fact that the integration testing is a wise-step process and thus, classes will be tested and integrated to form subsystems and then subsystems will be at the next step integrated in more important subsystems and so on till the integration of the whole system. Although, the justification is reasonable, it is surprising that on such small systems the genetic algorithm fails to reach always the best known solutions. We then decide to propose a memetic algorithm (MA), known to be more efficient than traditional genetic algorithms (Hoos et Sttzle, 2004). We also proposed a genetic algorithm used in a preliminary experiment to assess the efficiency of the proposed memetic algorithm and verify a need of using a more sophisticated algorithm to solve the CITO problem. We choose to propose another GA, an incremental GA instead of re-implementing the one proposed by Briand *et al.* (2002b) because, (i) the incremental GA is simpler to parametrize without lost of performance, and (ii) the GA proposed by Briand *et al.* (2002b) has been implemented through a commercial framework and we could not be sure that our re-implementation will faithfully represent theirs for a fair comparison. The details of the preliminary experiment and the results are detailed below after a brief description of the proposed incremental GA.

Proposed Genetic Algorithm

The proposed genetic algorithm is obtained by removing the local search operator in the memetic algorithm 5.3.2 and by adding a mutation operator. The mutation operator consists of a maximum of $muMvts$ moves performed randomly, the same move as in the local search heuristic described above. Therefore, the genetic algorithm has one more parameter: the intensity of the mutation $muMvts$.

Preliminary Experiment

Briand *et al.* (2002b) proposed a GA to solve the traditional CITO with the goal to minimize the stubbing cost. They defined in their experiment four cost functions to express the complexity of stubs to minimize: number of stubs (*i.e.*, number of broken dependencies), attribute coupling (AC), method coupling (MC), and weighted geometric average of both

attribute and method coupling (AMC) Briand *et al.* (2002b,a). We apply our algorithms using each of those cost functions on the five systems used in the experiment of Briand *et al.* (2002b) and also in two larger systems namely Ant 1.6.2 and Xerces 2.6.2. Table B.1 describes the systems used in this preliminary experiment giving the number of classes, associations, aggregations, inheritances, and cycles in their class diagrams, and the total lines of code (LOC) of the system. Table B.2 summarizes the parameters used for each algorithm in the experiment and the number of runs (nbRuns) performed in the experiment. Those parameters have been chosen after preliminary tests with different sets of parameters. We compare the results obtained by the two algorithms, the memetic algorithm (MA) and the genetic algorithm (IncGA) and refer also to the results obtained by the genetic algorithm (GA) proposed in (Briand *et al.*, 2002a) when it is possible. We summarize in Table B.3 the results obtained only with the cost function number of stubs.

On the small systems used in (Briand *et al.*, 2002b,a), MA and IncGA have the same performance in terms of best solutions found: both always reach the best known solution for all the five systems excepted IncGA once for Bcel 5.0. Results also show that MA and IncGA perform at least as better as the GA on those small systems but outperform the latter in many cases. Indeed, the GA proposed in (Briand *et al.*, 2002a) failed to find the best known solution for Bcel 5.0 and could not always find it for Ant and SPM. On the two larger systems, MA outperforms the IncGA that failed to reach the best known solution. Regarding the execution time to reach the best solution, Table B.3 shows that the IncGA requires in average much more time to reach the best solution than the MA. For example, MA requires in average 174 milliseconds to reach the best solution for Bcel 5.0 while IncGA requires 2,723 milliseconds. Unfortunately, we do not have the execution time of the algorithm proposed by Briand *et al.* (2002b) to make a fair comparison. However, the execution times required by IncGA and MA on the small systems is very negligible. Those results confirm the reported performance of MA over GA in the literature (Hoos et Sttzle, 2004; Moscato *et al.*, 2004). We observe similar trends with the other cost functions. We can then conclude that on small systems, the genetic algorithm is sufficient to find near to optimal solutions but on larger systems, a more powerful as the memetic algorithm is required. Our findings confirmed what is known in the search-based community (Hoos et Sttzle, 2004; Moscato *et al.*, 2004) as well in the software search-based community (Arcuri et Yao, 2007; Fraser *et al.*, 2015): *i.e.*, MA consistently performs better (*i.e.*, smaller result variability and faster convergence to a lower fitness value) with respect to GA.

Table B.1 Detailed Information of Analyzed Systems.

Classes	Associations	Aggregations	Inheritances	Cycles	Loc
Systems in Briand *et al.* (2002a)					
ATM: an automated teller machine simulation					
21	48	15	4	30	1390
Ant: a built tool for Java applications					
25	58	10	4	1178	1198
Dnsjava: an implementation of DNS in Java					
61	234	12	30	16	6710
Bcel: a library to analyze, create, and manipulate (binary) Java class					
45	244	4	46	416091	3033
SPM: a system for security zones and patrols monitoring					
19	70	2	11	654	4093
Large Systems					
Ant 1.6.2: a built tool for Java applications					
623	2136	36	393	x	178035
Xerces 2.0.1: A Java XML parser					
396	1058	3	220	x	120892

Table B.2 Parameters of Algorithms.

	Small Systems		Large Systems	
Parameter	IncGA	MA	IncGA	MA
$popSize$	100	50	200	100
$nbGens$	250000	100	20000000	20000
$muMvts$	1	-	1	-
nbRuns	100		20	

Table B.3 Performance of MA, IncGa, and GA, avg [min,max].

System	MA		IncGA		GA	
	Best Cost	Best Time(ms)	Best Cost	Best Time(ms)	Best Cost	Best Time(ms)
Small Systems in (Briand *et al.*, 2002b,a)						
Ant 0.0	10 [10, 10]	35 [33, 38]	10 [10, 10]	679 [270, 3000]	11 [10, 13]	NA
Atm 0.0	7 [7, 7]	27 [23, 38]	7 [7, 7]	170 [107, 289]	7 [7, 7]	NA
Bcel 5.0	60 [60, 60]	174 [162, 302]	60.03 [60, 63]	2723 [568, 15447]	65.53 [63, 70]	NA
Dnsjava 1.2	6 [6, 6]	249 [231, 354]	6 [6, 6]	858 [556, 1683]	6 [6, 6]	NA
Spm 0.0	16 [16, 16]	27 [23, 28]	16 [16, 16]	87 [58, 174]	16,76 [16, 20]	NA
Large Systems						
Ant 1.6.2	x	x	x	x	NA	NA
Xerces	x	x	x	x	NA	NA